What is OpenMP?

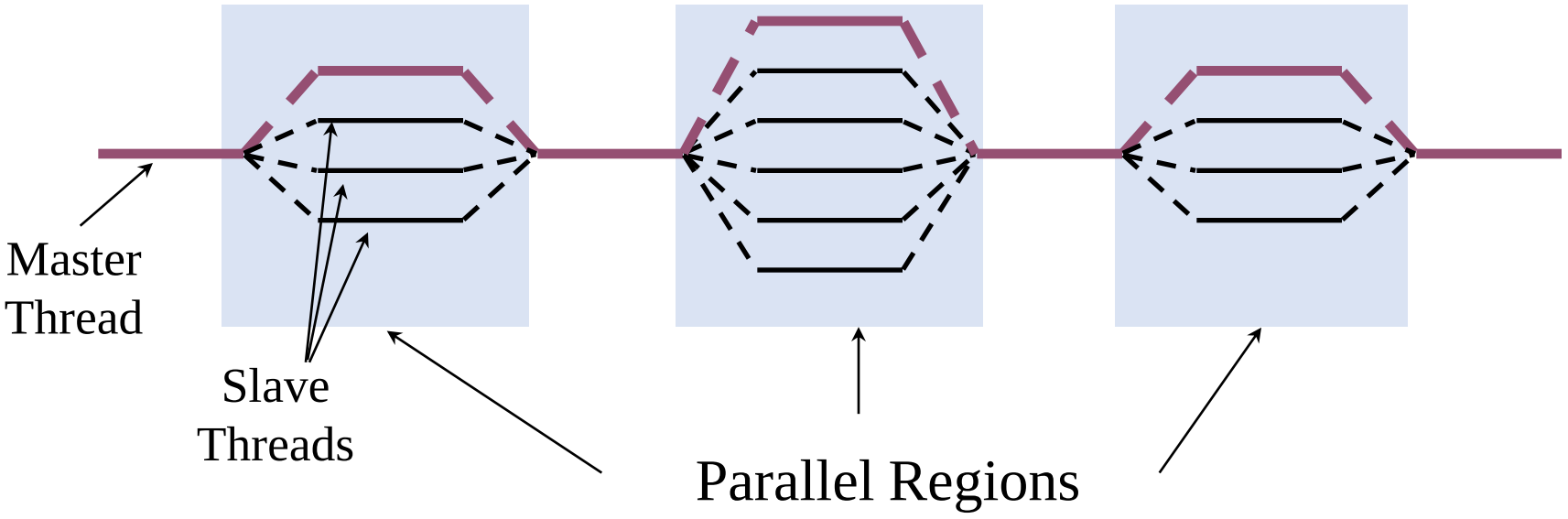
- **Parallel Computing** gives you more performance to throw at your problems.
  - Parallel computing is when a program uses concurrency to either:
    - Decrease the runtime for the solution to a problem.
    - Increase the size of the problem that can be solved
- **OpenMP** provides a **standard** for shared memory programming for scientific applications.
  - Has specific support for scientific application needs (*unlike Pthreads*).
  - Rapidly gaining acceptance among vendors and developers.
  - See *http://www.openmp.org* for more info.

OpenMP API Overview

- OpenMP - portable shared memory parallelism
  - An API for Writing Multithreaded Applications
    - API is a **set** of:
      - 1) compiler directives inserted in the source program
      - in addition to some 2) library functions and 3) environment variables
- OpenMP: Programming Model
  - **Fork-Join** Parallelism:
    - Master thread spawns/generates a team of threads as needed.
    - Parallelism is added incrementally:
      - i.e. the **sequential** program **evolves** into a **parallel** program

API Semantics

- Master thread executes sequential code.
- Master and slaves execute parallel code.
- **Note**:
  - very similar to fork-join semantics of *Pthreads* create/join primitives.



How is OpenMP typically used?

- OpenMP is usually used to parallelize loops:
  - Find your most **time-consuming loops**.
  - Split them up between threads.

Split-up this loop between multiple threads

```
void main()
{
  double Res[1000];

  for( int i=0;i<1000;i++ ) {
       do_huge_comp(Res[i]);
  }
}
```

Sequential Program

```
void main()
{
  double Res[1000];
  #pragma omp parallel for
  for( int i=0;i<1000;i++ ) {
     do_huge_comp(Res[i]);
  }
}
```

Parallel Program

OpenMP: How do threads interact?

- **OpenMP** is a **shared memory** model.
  - Threads communicate by sharing variables.
- Unintended sharing of data can lead to race conditions:
  - **race condition**: when the ***program's outcome*** changes as the threads are scheduled differently.
- To control race conditions:
  - Use **synchronization** to protect data conflicts.
- **Synchronization** is **expensive** so:
  - Often, we intend to change/control how data is stored to minimize the need for synchronization.

OpenMP Directives

- OpenMP implementation
  - Compiler directives, Library, and Environment, Unlike Pthreads (*purely a library*).

- Parallelization directives:
  - **`parallel region`**
  - **`parallel for`**
- Data environment directives:
  - **`shared`**, **`private`**, **`threadprivate`**, **`reduction`**, etc.
- Synchronization directives:
  - **`barrier`**, **`critical`**, **`atomic`**

- General Rules about Directives
  - They always apply to the *next statement* (or block of statements), which must be a structured block.

```
#pragma omp …
    Statement
#pragma omp …
{
    statement1;
    statement2;
    statement3;
}
```

OpenMP: Contents

- OpenMP's **constructs** fall into 5 categories:
  - Parallel Regions
  - Worksharing
  - Data Environment
  - Synchronization
  - Runtime functions/environment variables
  - Some Advanced Features

OpenMP Parallel Region

```
#pragma omp parallel
```

- A number of threads are spawned/created at the entry.
- Each thread executes the *same code* (SPMD model).
- The master thread *waits all other threads at the end*. (join)
- Very similar to a number of *create/join's* with the **same function** in *Pthreads*.

OpenMP: Parallel Regions

- You create threads in OpenMP with the "`omp parallel`" pragma.
- For example, to create a 4 thread Parallel Region:

Each thread *redundantly* executes the code within the structured block

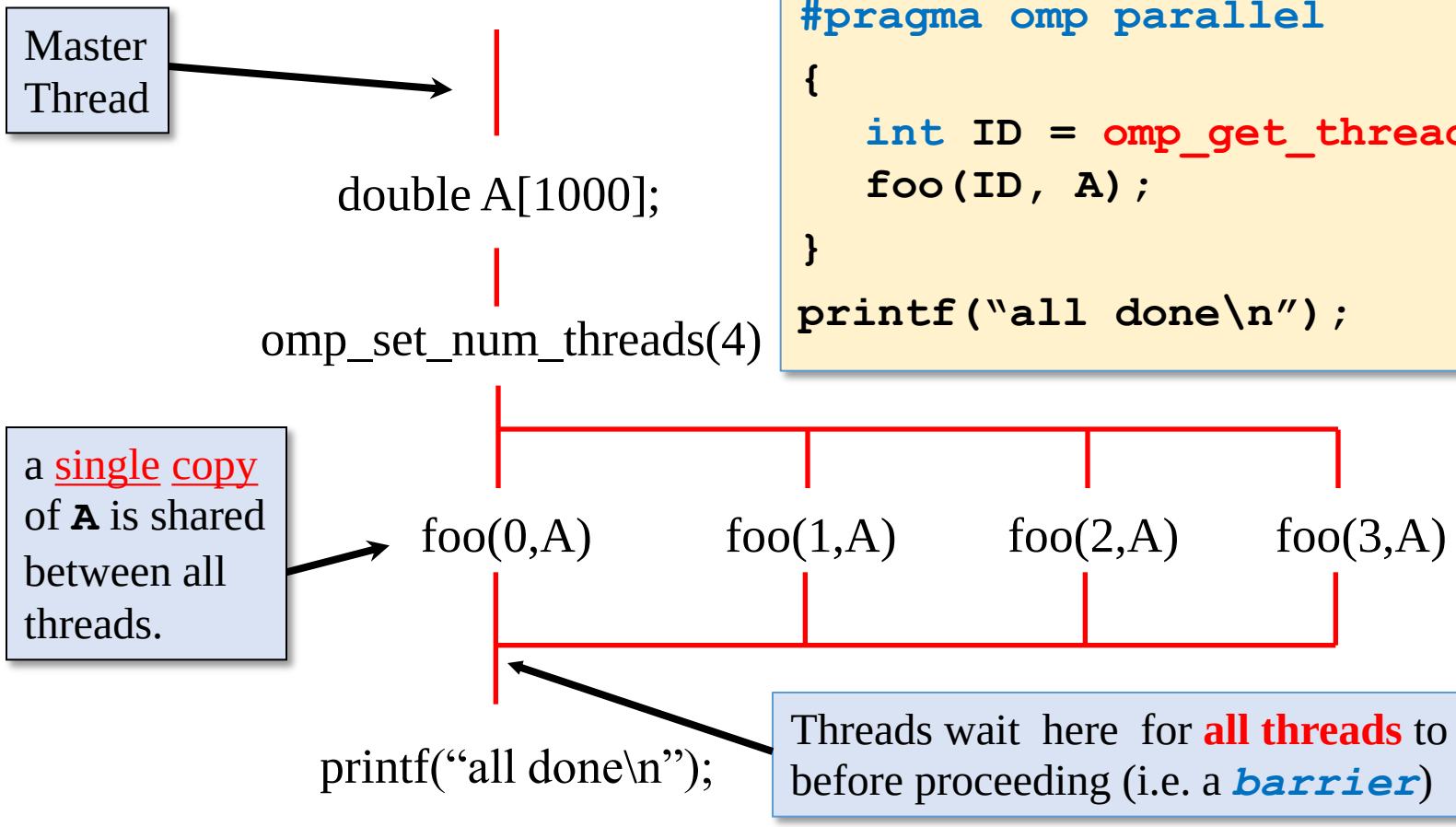
```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_thread_num();
    foo(ID,A);
}
```

Each thread calls `foo(ID,A)` for ID = 0 to 3

OpenMP: Parallel Regions

- Each thread executes the same code redundantly.

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
   int ID = omp_get_thread_num();
   foo(ID, A);
}
printf("all done\n");
```



#pragma omp parallel [clause list] Typical clauses in [clause list]

• Conditional parallelization

– if (scalar expression)

• Determine whether the parallel construct creates threads

• Degree of concurrency

– num_threads (integer expresson)

• number of threads to create

• Date Scoping

– private (variable list) • Specifies variables local to each thread

– firstprivate (variable list) • Similar to the private • Private variables are initialized to variable value before the parallel directive

– shared (variable list) • Specifies variables that are shared among all the threads

– default (data scoping specifier) • Default data scoping specifier may be shared or none

Example: #pragma omp parallel if (is_parallel == 1) num_threads(8) shared (var_b) private (var_a) firstprivate (var_c) default (none) { /* structured block */ }

- if (is_parallel == 1) num_threads(8) – If the value of the variable is_parallel is one, create 8 threads
- shared (var_b) – Each thread shares a single copy of variable b
- private (var_a) firstprivate (var_c) – Each thread gets private copies of variable var_a and var_c – Each private copy of var_c is initialized with the value of var_c in main thread when the parallel directive is encountered
- default (none) – Default state of a variable is specified as none (rather than shared) – Singals error if not all variables are specified as shared or private

OpenMP: Contents

- OpenMP's **constructs** fall into 5 categories:
  - Parallel Regions
  - Worksharing
  - Data Environment
  - Synchronization
  - Runtime functions/environment variables
  - Some Advanced Features

OpenMP: Work-Sharing Constructs

- The "**for**" Work-Sharing construct splits up loop iterations among the threads in a team

```
#pragma omp parallel
#pragma omp for
for ( I=0; I<N; I++ )
{
      NEAT_STUFF(I);
}
```

```
#pragma omp parallel for
for ( I=0; I<N; I++ )
{
      NEAT_STUFF(I);
}
```

… OR …

By default, there is a **barrier** at the end of the "**omp for**".

Use the "**nowait**" clause to **turn off (***disable it***)** the barrier.

Work Sharing Constructs

A motivating example

Sequential code

```
for(i=0;i<N;i++) { a[i]=a[i] + b[i]; }
```

OpenMP parallel region

```
#pragma omp parallel
{
  int id, i, Nthrds, istart, iend;
  id = omp_get_thread_num();
  Nthrds = omp_get_num_threads();
  istart = id * N / Nthrds;
  iend = (id+1) * N / Nthrds;
  for(i=istart; i<iend; i++){
     a[i] = a[i] + b[i];}
}
```

OpenMP parallel region and a work-sharing for-construct

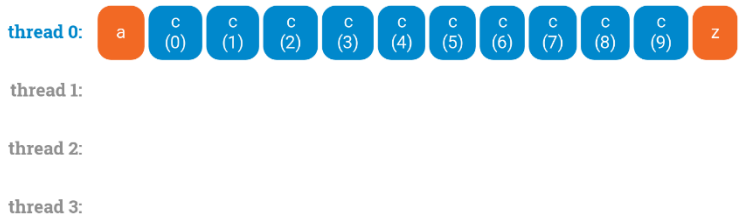
```
#pragma omp parallel
#pragma omp for schedule(static)
for(i=0;i<N;i++) { a[i] = a[i] + b[i];}
```

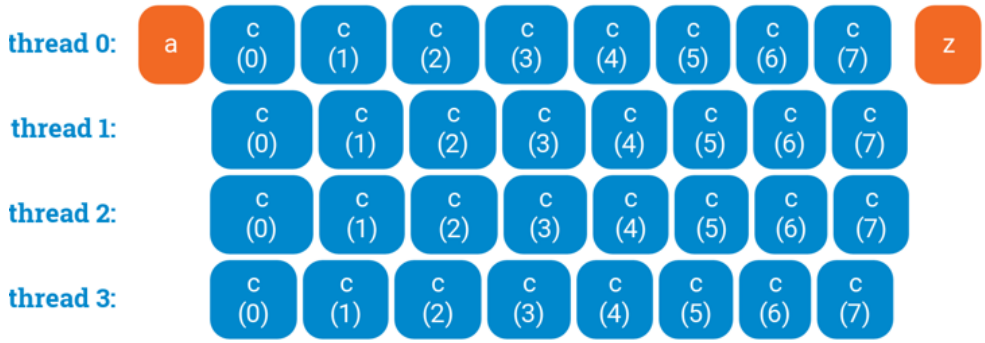
OpenMP Directive: parallel for

```
a();

for(int i=0; i<10; ++i)
{
        c(i);
}
z();
```

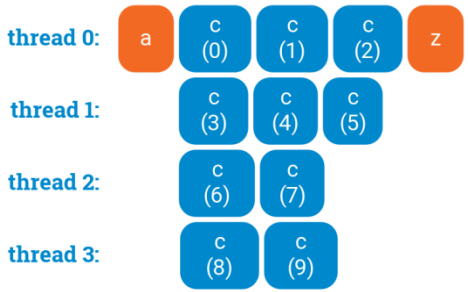
```
omp_set_num_threads(4);
a();
#pragma omp parallel
for( int i = 0; i < 8; ++i)
{
     c(i);
}
z();
```





OpenMP Directive: parallel for

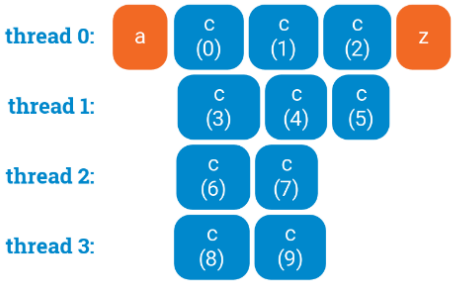
```
omp_set_num_threads(4);
a();
#pragma omp parallel
{
    #pragma omp for
    for( int i= 0; i<10; ++i)
    {
        c(i);
    }
}
z();
```




```
omp_set_num_threads(4);
a();

#pragma omp parallel for
for( int i=0; i<10; ++i)
{
        c(i);
}

z();
```



It is just a shorthand

Work Sharing Directives

- Always occur within a parallel region directive
- Two principal:
  - parallel for
  - parallel section

OpenMP Parallel For
-- Each thread executes a subset of the iterations
-- All threads wait at the end of the parallel for

```
#pragma omp parallel
#pragma omp for
for( … ) { … }
```

is equivalent to

```
#pragma omp parallel for
for ( … ) { … }
```

Note that: Same for `parallel sections`

```
#pragma omp parallel
{
  #pragma omp sections
  {
      ……
      #pragma omp section
      { …… }
      #pragma omp section
      { …… }
       ……
  }
}
```

Example: Matrix Multiply

Sequential Approach

```
for( i=0; i<n; i++ )
   for( j=0; j<n; j++ ) {
       c[i][j] = 0.0;
       for( k=0; k<n; k++ )
            c[i][j] += a[i][k] * b[k][j];
   }
```

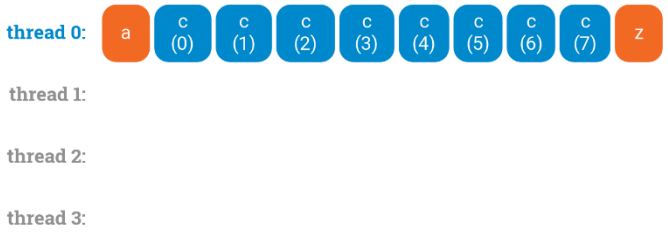
OpenMP Based Parallel Approach

```
#pragma omp parallel for
for( i=0; i<n; i++ )
   for( j=0; j<n; j++ ) {
       c[i][j] = 0.0;
       for( k=0; k<n; k++ )
            c[i][j] += a[i][k] * b[k][j];
   }
```

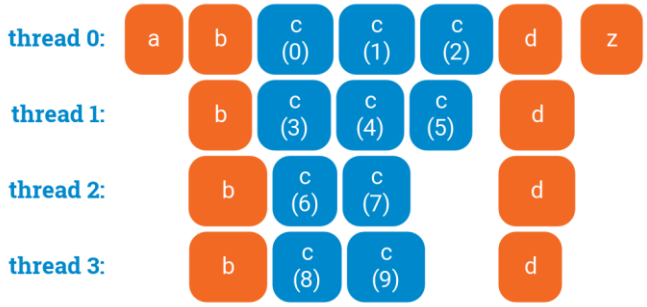
OpenMP Directive: parallel for

Common mistakes in the use `omp parallel or omp for`

```
a();
#pragma omp for
for (int i=0; i<8; ++i)
{
    c(i);
}
z();
```




```
a();
#pragma omp parallel
{
    b();
    #pragma omp for
    for (int i=0; i<10; ++i) {
        c(i);
    }
    d();
}
z();
```



OpenMP **parallel for**

Waiting / No Waiting

Multiple Work Sharing Directives

May occur within a **single parallel region**

All threads **wait at the end** of the **first for**.

```
#pragma omp parallel
{
        ……
    #pragma omp for
    for( ; ; ) { … }
        ……
    #pragma omp for
    for( ; ; ) { … }
        ……
}
```

The `nowait` Qualifier

Threads proceed to second **for** without waiting.

```
#pragma omp parallel
{
    #pragma omp for nowait
    for( ; ; ) { … }
    #pragma omp for
    for( ; ; ) { … }
}
```

Note the Difference between ...

```
#pragma omp parallel
{
    #pragma omp for
    for( ; ; ) { … }
    foo();
    #pragma omp for
    for( ; ; ) { … }
}
```

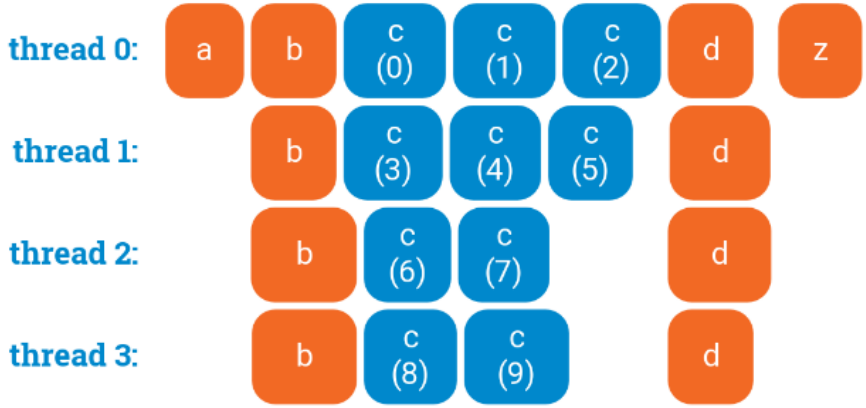
… and ...

```
#pragma omp parallel for
for( ; ; ) { … }
foo();
#pragma omp parallel for
for( ; ; ) { … }
```

OpenMP parallel for loops: waiting

- In a `parallel` region, OpenMP will automatically wait for all threads to finish before execution continues.
- There is also a *synchronization point* after each `omp for` loop;
  - here no thread will execute `d()` until all threads are done with the loop

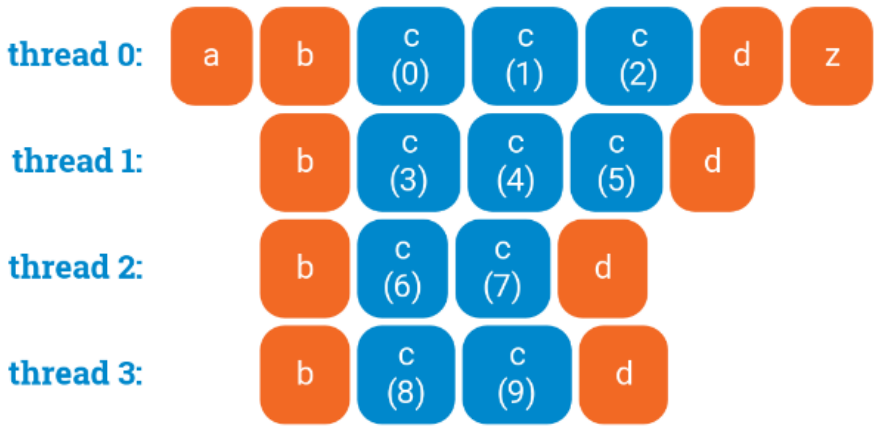
```
a();
#pragma omp parallel
{
  b();
  #pragma omp for
  for (int i=0; i<10; ++i)
  {
       c(i);
  }
  d();
}
z();
```



OpenMP parallel for loops: waiting

- However, if you do not need synchronization after the loop, you can disable it with nowait:

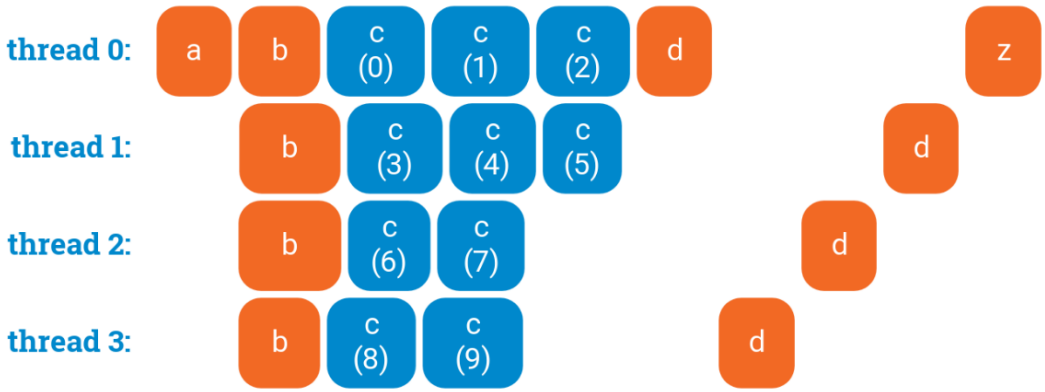
```
a();
#pragma omp parallel
{
  b();
  #pragma omp for nowait
  for (int i=0; i<10; ++i)
  {
       c(i);
  }
  d();
}
z();
```



OpenMP parallel for loops:
Interaction with critical sections

If you need a critical section after a loop, note that normally OpenMP will first wait for all threads to finish their loop iterations before letting any of the threads to enter a critical section:

```
a();
#pragma omp parallel
{
  b();
  #pragma omp for
  for (int i=0; i<10; ++i)
  {
     c(i);
  }
  #pragma omp critical
  {
    d();
  }
}
z();
```

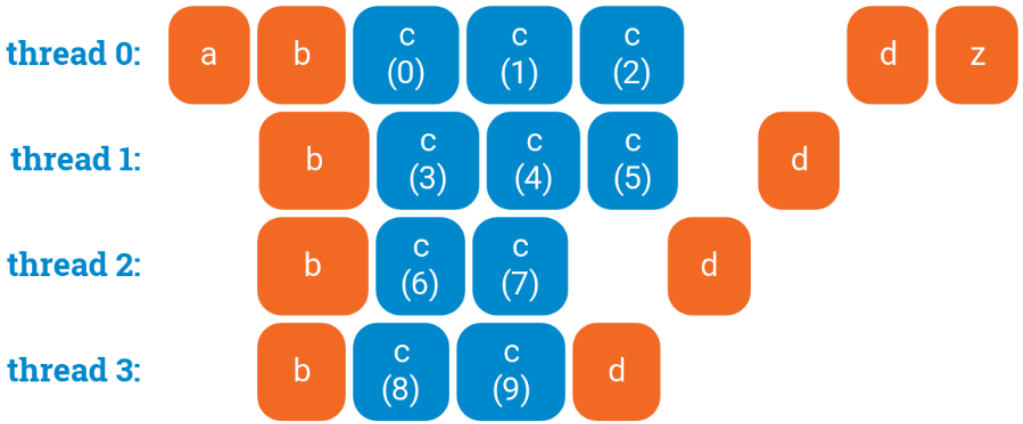


OpenMP parallel for loops:
Interaction with critical sections

You can disable waiting, so that some threads can start doing post-processing early. This would make sense if, e.g., **d()** updates some global data structure based on what the thread computed in its own part of the parallel for loop:

```
a();
#pragma omp parallel
{
  b();
  #pragma omp for nowait
  for (int i=0; i<10; ++i)
  {
     c(i);
  }
  #pragma omp critical
  {
    d();
  }
}
z();
```

Notice



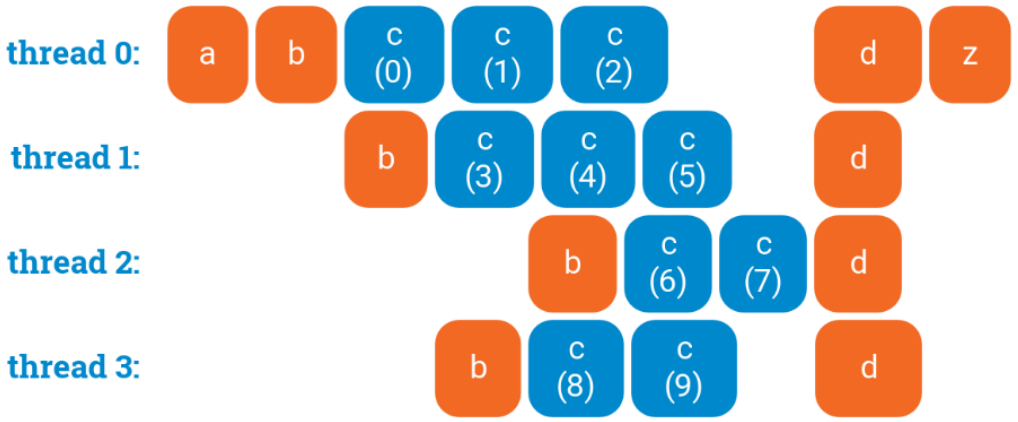
OpenMP parallel for loops:

No waiting before a loop

Now, note that there is no synchronization point *before the loop starts*.

If threads reach the for loop at different times, they can start their own part of the work as soon as they are there, without waiting for the other threads:

```
a();
#pragma omp parallel
{
  #pragma omp critical
  {
    b();
  }
  #pragma omp for
  for (int i=0; i<10; ++i)
  {
    c(i);
  }
  d();
}
z();
```



References

- OpenMP topic: Loop parallelism
  - https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-loop.html
- A “Hands-on” Introduction to OpenMP
  - https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf
- OpenMP in a nutshell
  - http://www.bowdoin.edu/~ltoma/teaching/cs3225-GIS/fall17/Lectures/openmp.html
- #pragma omp parallel (IBM)
  - https://www.ibm.com/support/knowledgecenter/SSGH3R_13.1.3/com.ibm.xlcpp1313.aix.doc/compiler_ref/prag_omp_parallel.html
- OpenMP Directives (Microsoft)
  - https://docs.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-directives?view=vs-2019
- Guide into OpenMP: Easy multithreading programming for C++
  - https://bisqwit.iki.fi/story/howto/openmp/

Open MP

Assigning Iterations to Threads

- Schedule clause of the for directive deals with the assignment of iterations to threads.
- Schedule directive is schedule(scheduling_class[, parameter]).
- OpenMP supports four scheduling classes:
  - static
  - Dynamic
  - guided
  - runtime

Assigning Iterations to Threads

- Static N Schedule(interleaved) (Interleaved)
- The iteration space is broken in chunks of approximately size
- *N/num − threads*. Then these chunks are assigned to the threads in a Round-Robin fashion

- Characteristics of the static schedules
  - Low overhead
  - Good locality (usually)
  - Can have load imbalance problems

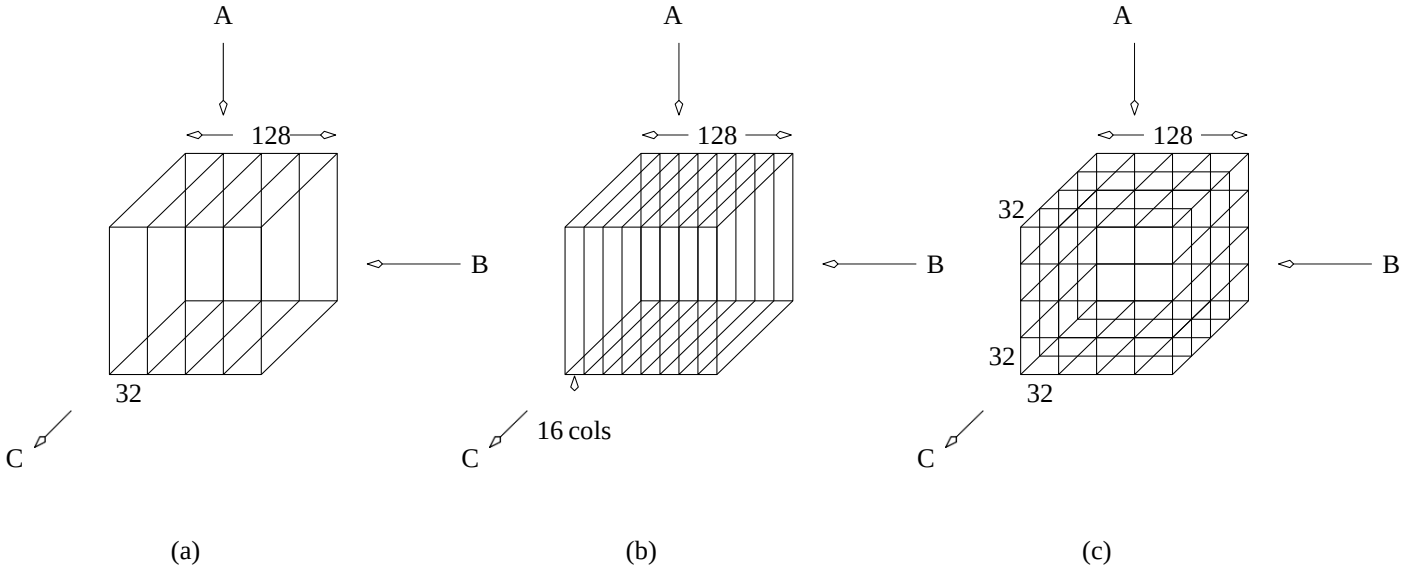
Assigning Iterations to Threads

```
/* static scheduling of matrix multiplication loops */
#pragma omp parallel default(private) shared (a, b, c, dim) \
                    num_threads(4)

 #pragma omp for schedule(static)
 for (i = 0; i < dim; i++) {
     for (j = 0; j < dim; j++) {

           c(i,j) = 0;
           for (k = 0; k < dim; k++) {
               c(i,j) += a(i, k) * b(k, j);
           }
       }
   }
```

Assigning Iterations to Threads: Example



Three different schedules using the static scheduling class of OpenMP.

Assigning Iterations to Threads: Example

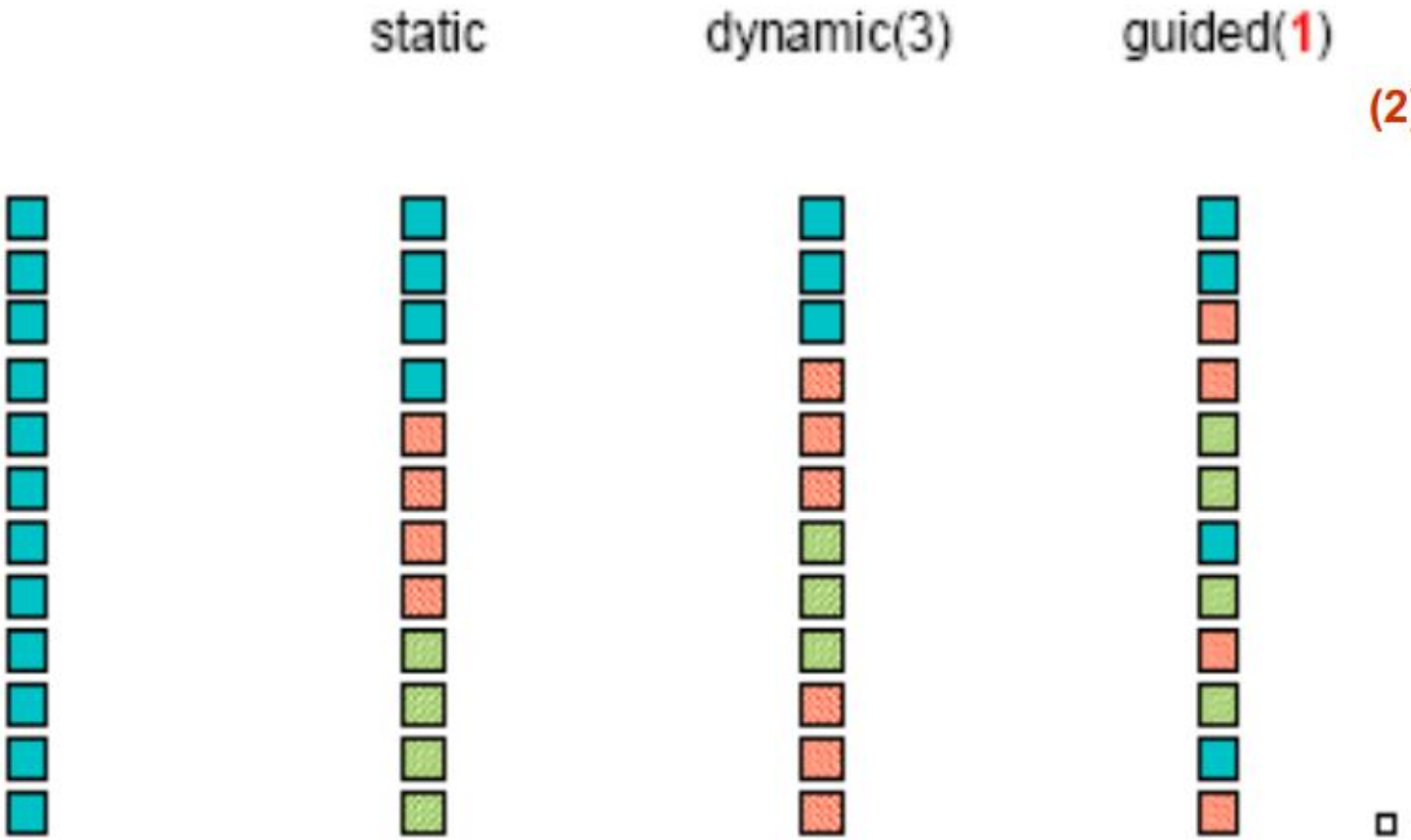
- Dynamic, N schedule
- Threads dynamically grab chunks of *N* iterations until all iterations have been executed. If no chunk is specified, *N* = 1.

- Guided, N schedule
- Variant of **`dynamic`**. The size of the chunks deceases as the threads grab iterations, but it is at least of size *N*. If no chunk is specified,
- *N* = 1.

- Characteristics of dynamic schedules
  - Higher overhead
  - Not very good locality (usually)
    Can solve imbalance problems

- Runtime, N Schedule
- it is desirable to delay scheduling decisions until runtime.

Assigning Iterations to Threads



Parallel For Loops

- Often, it is desirable to have a sequence of `for`-directives within a parallel construct that do not execute an implicit barrier at the end of each `for` directive.

- OpenMP provides a clause – `nowait`, which can be used with a `for` directive.

Parallel For Loops: Example

```
#pragma omp parallel
{
    #pragma omp for nowait
        for (i = 0; i < nmax; i++)
            if (isEqual(name,
                current_list[i])
                processCurrentName(name);
    #pragma omp for
        for (i = 0; i < mmax; i++)
            if (isEqual(name,
                past_list[i])
                processPastName(name);

 }
```

The `sections` Directive

- OpenMP supports non-iterative parallel task assignment using the `sections` directive.
- The general form of the `sections` directive is as follows:

```
#pragma omp sections [clause list]
{
  [#pragma omp section
    /* structured block */
  ]
  [#pragma omp section
    /* structured block */
  ]
  ...
}
```

The `sections` Directive: Example

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            taskA();
        }
        #pragma omp section
        {
            taskB();
        }
        #pragma omp section
        {
            taskC();
        }
    }
}
```

Merging Directives

```
1.#pragma omp parallel default (private) shared (n)
2 {
3 #pragma omp for
4 for (i = 0 < i < n; i++) {
5 /* body of parallel for loop */
6 }
7 }
```

Merging Directives

Identical to

```
1 #pragma omp parallel for default (private) shared (n)
2 {
3 for (i = 0 < i < n; i++) {
4 /* body of parallel for loop */
5 }
6 }
```

Merging Directives

```
1 #pragma omp parallel
2 {
3 #pragma omp sections
4 {
5 #pragma omp section
6 {
7 taskA();
8 }
9 #pragma omp section
10 {
11 taskB();
12 }
13 /* other sections here */
14 }
15 }
```

Merging Directives

```
1 #pragma omp parallel sections
2 {
3 #pragma omp section
4 {
5  taskA();
6 }
 7  #pragma omp section
8 {
9 taskB();
10 }
11 /* other sections here */
12 }
```

Nesting parallel Directives

```
#pragma omp parallel for default(private) shared (a, b, c, dim) \
2 num_threads(2)
3 for (i = 0; i < dim; i++) {
4 #pragma omp parallel for default(private) shared (a, b, c, dim) \
 5 num_threads(2)
 6 for (j = 0; j < dim; j++) {
 7 c(i,j) = 0;
 8 #pragma omp parallel for default(private) \
 9 shared (a, b, c, dim) num_threads(2)
10 for (k = 0; k < dim; k++) {
11 c(i,j) += a(i, k) * b(k, j);
 12 }
13 }
14 }
```

Synchronization Constructs in OpenMP

- Synchronization Point: The barrier Directive

#pragma omp barrier

- all threads in a team wait until others have caught up, and then release.
- nested parallel directives, the barrier directive binds to the closest parallel directive.

Single Thread Executions: The single and master Directives

- #pragma omp single [clause list]

/* structured block */

- A single directive specifies a structured block that is executed by a single (arbitrary) thread.
- clauses list can be private, firstprivate, and nowait

Single Thread Executions: The single and master Directives

- The master directive is a specialization of the single directive in which only the master thread executes the structured block.
- The syntax of the master directive is as follows:
- #pragma omp master

  structured block

Critical Sections: The critical and atomic Directives

- Critical Sections: The critical and atomic directives
- #pragma omp critical [(name)]

  structured block
- name can be used to identify a critical region.
- The use of name allows different threads to execute different code while being protected from each other.

Critical Sections: The critical and atomic Directives

- Critical Sections: The critical and atomic directives
- #pragma omp critical [(name)]
  structured block
- Critical Directive: allows only one thread is inside the critical region. All the others must wait.
- name can be used to identify a critical region.
- name field is optional. If no name is specified, the critical section maps to a default name that is the same for all unnamed critical sections
- The use of name allows different threads to execute different code while being protected from each other.

Critical Sections: The critical and atomic Directives

```
#pragma omp parallel sections
 {
 #pragma parallel section
 {
 /* producer thread */
 task = produce_task();
 #pragma omp critical ( task_queue)
 {
 insert_into_queue(task);
 }
```

Critical Sections: The critical and atomic Directives

```
}
 #pragma parallel section
{
 /* consumer thread */
 #pragma omp critical ( task_queue)
 {
 task = extract_from_queue(task);
 }
 consume_task(task);
 }
 }
```

Note : queue full and queue empty conditions must be explicitly handled here in functions insert_into_queue and extract_from_queue.

Critical Sections: The critical and atomic Directives

- The atomic directive specifies that the memory location update in the following instruction should be performed as an atomic operation.
- The update instruction can be one of the following forms:

  x binary_operation = expr

  x++

  ++x

  x--

  --x

Ordered Directive

- Enclosed code is executed in the same order as would occur in sequential execution of the loop
- #pragma omp ordered

  structured block

Ordered Directive

```
cumul_sum[0] = list[0];
 #pragma omp parallel for private (i) \
shared (cumul_sum, list, n) ordered
for (i = 1; i < n; i++)
{
 /* other processing on list[i] if needed */
 #pragma omp ordered
 {
 cumul_sum[i] = cumul_sum[i-1] + list[i];
 }
 }
```

Memory Consistency: The flush Directive

- synchronization point at which the system must provide a consistent view of memory
  - all thread visible variables must be written back to memory (if no list is provided), otherwise only those in the list are written back
- Implicit flushes of all variables occur automatically at
  - all explicit and implicit barriers
  - entry and exit from critical regions
  - entry and exit from lock routines
- Directives
  - #pragma omp flush [(list)]

Data Handling in OpenMP

- #pragma omp parallel[data scope clauses ...]
  - Shared
  - private
  - firstprivate
  - lastprivate
  - threadprivate
  - default

Shared

- Shared data among team of threads
- Each thread can modify shared variables
- Data corruption is possible when multiple threads attempt to update the same memory location
- Data correctness is user's responsibility

```
float dot_prod(float* a, float* b, int N)
{
float sum = 0. 0 ;
 #pragma omp parallel for shared(sum)
for(int i=0; i<n; i++)
sum+=a[i]*b[i];
}
return sum
}
```

Example

```
int x=1;
#pragmaompparallelshared    (x) num_threads(2)
{
    x++;
    printf("%d\n",x);
}
printf("%d\n",x);
```

Example

```
int x=1;
#pragmaompparallelshared    (x) num_threads(2)
{
    x++;
    printf("%d\n",x);
}
printf("%d\n",x);
```

Prints 2 or 3 (three printfs in total)

Private

- The values of private data are undefined upon entry to and exit from the specific construct. Loop iteration variable is private by default Example:

```
   void* work(float* c, int N)
{
 float x, y;
int i;
#pragma omp parallel for private(x,y)
for(i=0; i<N; i++)
X=a[i]; y=b[i];
C[i]=x+y;
}
}
```

Example

```
int x=1;
#pragmaomppparallelprivate    (x) num_threads(2)
{
    x++;
    printf("%d\n",x);
}
printf("%d\n",x);
```

Can print anything (twice, same or different)

Example

```
int x=1;
#pragmaompparallelprivate    (x) num_threads(2)
{
    x++;
    printf("%d\n",x);
}
printf("%d\n",x);
```

Prints 1

firstprivate clause(Data Scope)

- The clause combines behavior of private clause with automatic initialization of the variables in its list with values prior to parallel region.

Example:

```
int b=51, n=100 ;
printf(“Before parallel loop: b=%d ,n=%d\n”,b,n) #pragma omp parallel for private(i), firstprivate(b)
for(i=0;i<n;i++)
{
a[i]=i+b;
}
```

firstprivate

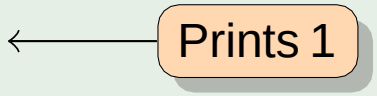
- incr=0;
- #pragma omp parallel for firstprivate(incr);
- for (I=0;I<=MAX;I++)
- {
- if ((I%2)==0) incr++;
- A(I)=incr;
- }

Example

```
int x=1;
#pragmaompparallelfirstprivate    (x) num_threads(2)
{
    x++;
    printf("%d\n",x);
}
printf("%d\n",x);
```

Example

```
int x=1;
#pragmaompparallelfirstprivate    (x) num_threads(2)
{
    x++;
    printf("%d\n",x);
}
printf("%d\n",x);
```



Lastprivate

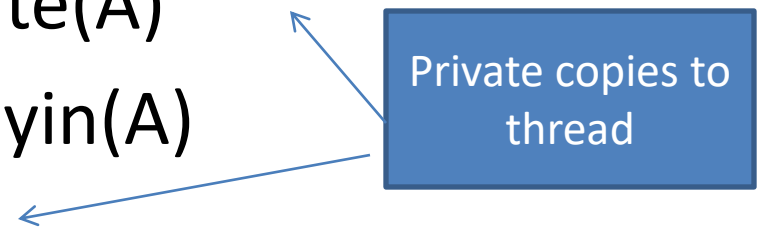
- Variables update shared variable using value from last iteration

```
void sq2(int n, double *last term)
{
double x; int i;
#pragma omp parallel
#pragma omp for lastprivate(x)
for ( i = 0; i < n; i++)
{
x = a[i]*a[i] + b[i]*b[i];
b[i] = sqrt( x);
}
last term = x;
 }
```

Threadprivate and copyin clause

- Preserves global scope for per-thread storage
- Legal for name-space-scope and file-scope
- Use copyin to initialize from master thread

```
struct A;
#pragma omp threadprivate(A)
#pragma omp parallel copyin(A)
do_something_to(&A)
#pragma omp paralle l do_something_else_to(&A)
```



Example

```
char *foo ()
{
   static char buffer[BUF_SIZE];
#pragma omp threadprivate(buffer) /* Creates one static copy of buffer per thread*/


   ...

   return buffer;
}
```

Example

```
char foo ()
{
  static char buffer[BUF_SIZE];
  #pragma omp threadprivate(buffer)

  ...

  return buffer;
}
```

Now foo can be called by multiple threads at the same time

foo returns correct address to caller

OpenMP Library Functions

In addition to directives, OpenMP also supports a number of functions that allow a programmer to control the execution of threaded programs.

Thread control:

- `void omp_set_num_threads(int number)` — Set the `number` of threads to be used for subsequent parallel regions (without a `num_threads` clause)
- `int omp_get_thread_num(void)` — Get my thread ID in the current team
- `int omp_get_num_threads(void)` — Get the number of threads in the team that currently executes the region
- `int omp_get_max_threads(void)` — Get the maximum number of threads that could be used to form a new team in a parallel region without a num_threads clause

Processor level:

- `int omp_get_num_procs(void)` — Get the number of processors available on the device

Check if inside a parallel region:

- `int omp_in_parallel(void)` — If execution is inside an active parallel region, the returned value evaluates to *true*

You can change the scheduling policy in your code for places where you have used the `schedule(runtime)` clause:

Schedule control:

- `void omp_set_schedule(omp_sched_t kind, int chunk_size)`
- `void omp_get_schedule(omp_sched_t* kind, int* chunk_size)`

```
1 typedef enum omp_sched_t {
2     omp_sched_static  = 1,
3     omp_sched_dynamic = 2,
4     omp_sched_guided  = 3,
5     omp_sched_auto    = 4
6 } omp_sched_t;
```

Defined in the header `omp.h`

```
1  #include <omp.h>
2
3  int main(int argc, char* argv[])
4  {
5  #pragma omp parallel
6      {
7          // set a desired schedule.  This will overwrite what has been set in
8          // the OMP_SCHEDULE environment variable
9          omp_set_schedule(omp_sched_dynamic, 1);
10
11         // test it
12         omp_sched_t sched;
13         int chunk;
14         omp_get_schedule(&sched, &chunk);
15 #pragma omp master
16         cout << "schedule = " << sched << ", chunk = " << chunk << endl;
17
18 #pragma omp for schedule(runtime)
19         for (int i = 0; i < N; ++i)
20             // do work
```

OpenMP provides convenient timing routines:

Timer:

- `double omp_get_wtime(void)`
- `double omp_get_wtick(void)`

Returns the per-thread wall time between two measurement points

Returns the number of seconds between two consecutive clock cycles.

```
1  int main(int argc, char* argv[])
2  {
3  #pragma omp parallel
4      {
5          const int tid = omp_get_thread_num();
6          random_device rd; // #include <random>
7          mt19937 gen(rd());
8          uniform_int_distribution<> uniform(1, 500);
9
10         const int mytime = uniform(gen) * 1000; // micro seconds
11 #pragma omp critical
12         cout << "thread " << tid << ": " << mytime*1.0e-6 << " sec (expect)" << endl;
13
14         const double start = omp_get_wtime();
15         usleep(mytime); // #include <unistd.h>
16         const double duration = omp_get_wtime() - start;
17 #pragma omp critical
18         cout << "thread " << tid << ": " << mytime*1.0e-6 << " sec (measured)" << endl;
19     }
20     return 0;
21 }
```

Output:

```
1 thread 1: 0.031 sec (expect)
2 thread 3: 0.204 sec (expect)
3 thread 2: 0.269 sec (expect)
4 thread 0: 0.382 sec (expect)
5 thread 1: 0.031 sec (measured)
6 thread 3: 0.204 sec (measured)
7 thread 2: 0.269 sec (measured)
8 thread 0: 0.382 sec (measured)
```

OpenMP Library Functions

```
/* controlling and monitoring thread creation
*/  void omp_set_dynamic (int dynamic_threads);
int omp_get_dynamic ();
void omp_set_nested (int nested);
int omp_get_nested ();

/* mutual exclusion */
void omp_init_lock (omp_lock_t *lock);
void omp_destroy_lock (omp_lock_t
void *lock);  omp_set_lock (omp_lock_t
void *lock);  omp_unset_lock (omp_lock_t
int omp_test_lock (omp_lock_t *lock);
    *lock);
```

In addition, all lock routines also have a nested lock counterpart for recursive mutexes.

Lock routines in OpenMP:

Simple locks:

- `void omp_init_lock(omp_lock_t* mutex)`
- `void omp_destroy_lock(omp_lock_t* mutex)`
- `void omp_set_lock(omp_lock_t* mutex)`
- `void omp_unset_lock(omp_lock_t* mutex)`

Initialize lock to unlocked state. Must be called before the lock can be used.

Ensure the lock is uninitialized. Must be called when the lock is not used anymore

Acquire the lock

Release the lock

```
1 #include <omp.h>
2
3 int main(int argc, char* argv[])
4 {
5     omp_lock_t mutex;
6     omp_init_lock(&mutex);
7 #pragma omp parallel
8     {
9         omp_set_lock(&mutex);
10        // critical section
11        omp_unset_lock(&mutex);
12    }
13    omp_destroy_lock(&mutex);
14    return 0;
15 }
```

Mutual exclusion using a lock

Mutual exclusion using a `critical` construct

```
1 int main(int argc, char* argv[])
2 {
3 #pragma omp parallel
4     {
5 #pragma omp critical
6         {
7             // critical section
8         }
9     }
10    return 0;
11 }
```

Environment Variables in OpenMP

- `OMP NUM THREADS`: This environment variable specifies the default number of threads created upon entering a `parallel` region.
- `OMP SET DYNAMIC`: Determines if the number of threads can be dynamically changed.
- `OMP NESTED`: Turns on nested parallelism.
- `OMP SCHEDULE`: Scheduling of for-loops if the clause specifies runtime.

Note: The name of the environment variable must be upper case. Assigned values are case-insensitive and may have leading and/or trailing white space

OpenMP behavior:

- `OMP_DYNAMIC='true' or 'FALSE'`
- `OMP_NESTED='True' or ' false '`

Set dynamic mode. Its default is implementation defined. If dynamic adjustment of the number of threads is supported it will be *true*, otherwise *false*.

Set nested parallelism. Its default is implementation defined. If nested parallelism is supported it will be *true*, otherwise *false*.

Schedule control:

- `OMP_SCHEDULE='schedule[, chunk_size]'`

Set the scheduling policy for the **`schedule`**`(runtime)` clause. The `omp_set_schedule()` API call can overwrite the value at runtime.

Thread control:

- `OMP_NUM_THREADS=positive_integer`

Set the maximum number of threads. The value of this environment variable can be obtained with the `omp_get_max_threads()` API call at runtime. The **`num_threads`**`(n)` clause overwrites the value for the given region. If dynamic mode is enabled, your code may run with fewer threads depending on available system resources.

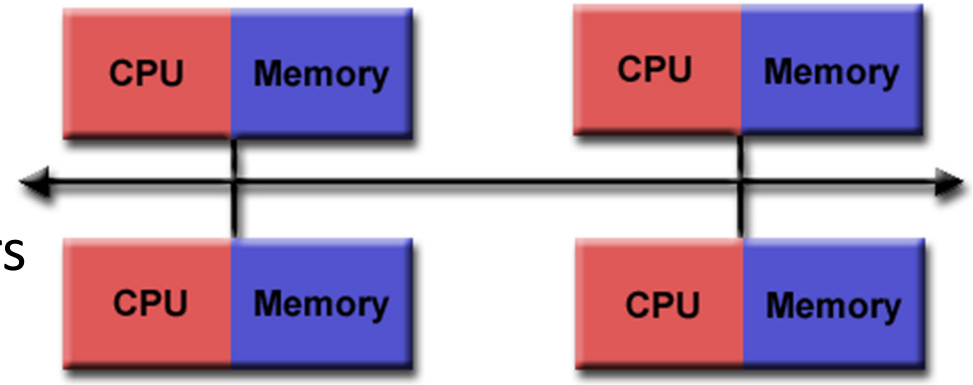
CSE 613: Parallel Programming

Lecture 14
(The Message Passing Interface)

Rezaul A. Chowdhury
Department of Computer Science
SUNY Stony Brook
Spring 2012

Principles of Message-Passing Programming

- One of the oldest and most widely used approaches for programming parallel computers



Source: Blaise Barney, LLNL

- Two key attributes
  - Assumes a partitioned address space
  - Supports only explicit parallelism
- Two immediate implications of partitioned address space
  - Data must be explicitly partitioned and placed to appropriate partitions
  - Each interaction (read-only and read/write) requires cooperation between two processes: process that has the data, and the one that wants to access the data

Structure of Message-Passing Programs

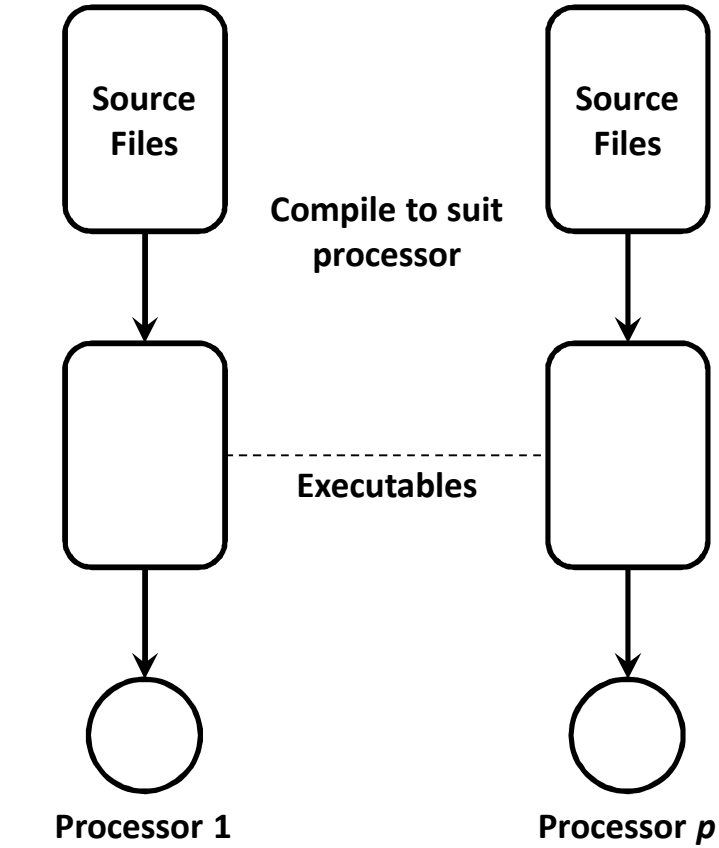
Asynchronous

- All concurrent tasks execute asynchronously
- Most general (can implement any parallel algorithm)
- Can be difficult to reason about
- Can have non-deterministic behavior due to races

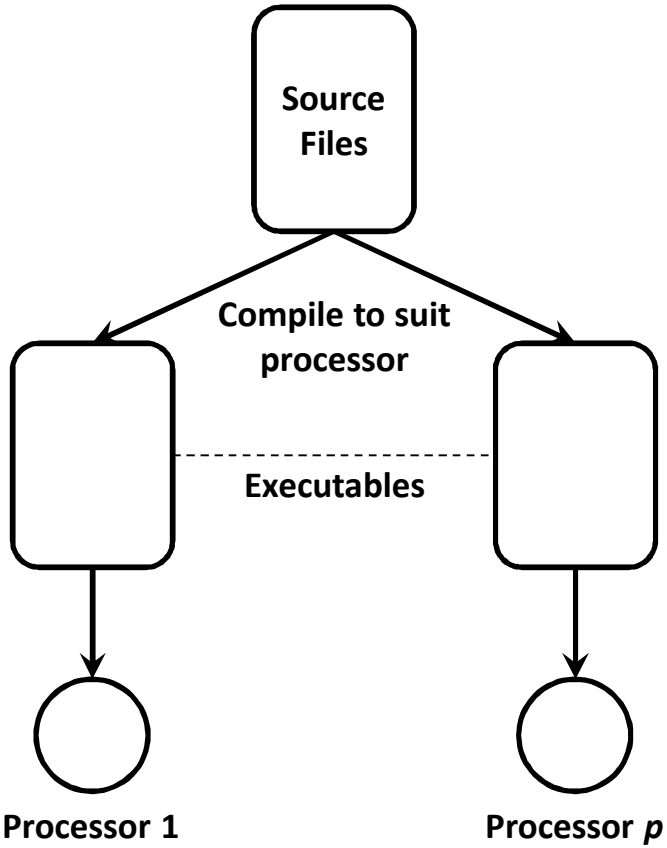
Loosly Synchronous

- A good compromise between synchronous and asynchronous
- Tasks or subset of tasks synchronize to interact
- Between the interactions tasks execute asynchronously
- Easy to reason about these programs

Structure of Message-Passing Programs



Multiple Program Multiple Data (MPMD)



Single Program Multiple Data (SPMD)

- Ultimate flexibility in parallel programming
- Unscalable

- Most message-passing programs
- Loosely synchronous or completely asynchronous

The Building Blocks: Send & Receive Operations

send(&*data*, *n*, *dest* **):**

Send *n* items pointed to by &*data* to a processor with id *dest*

receive(&*data*, *n*, *src* **):**

Receive *n* items from a processor with id *src* to location pointed to by &*data*

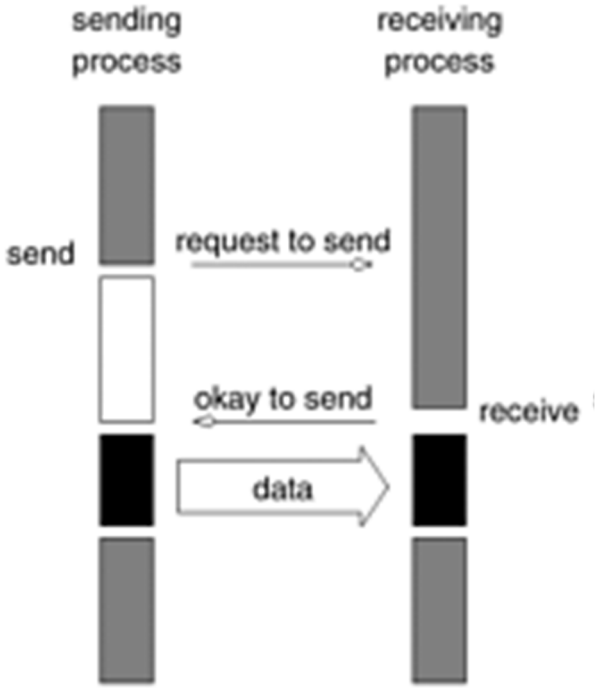
But wait! What P1 prints when P0 and P1 execute the following code?

```
1          P0                                P1
2
3          a = 100;                          receive(&a, 1, 0)
4          send(&a, 1, 1);                   printf("%d\n", a);
5          a=0;
```

Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

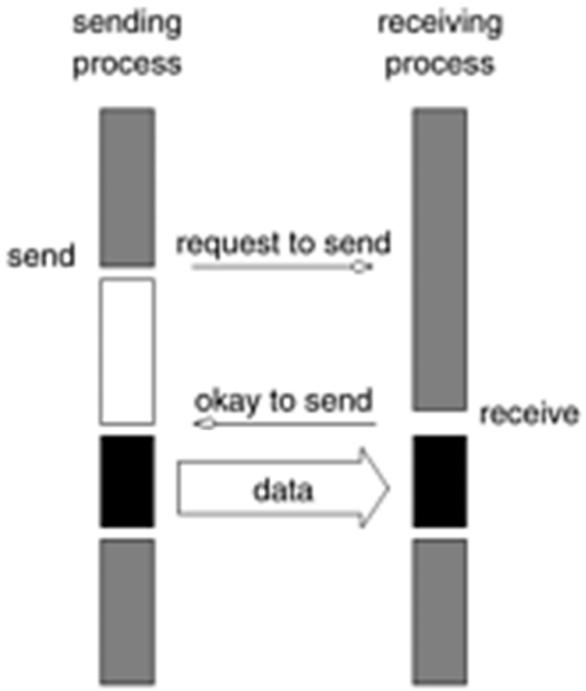
Blocking Non-Buffered Send / Receive

Sending operation waits until the matching receive operation is encountered at the receiving process, and data transfer is complete.



Blocking Non-Buffered Send / Receive

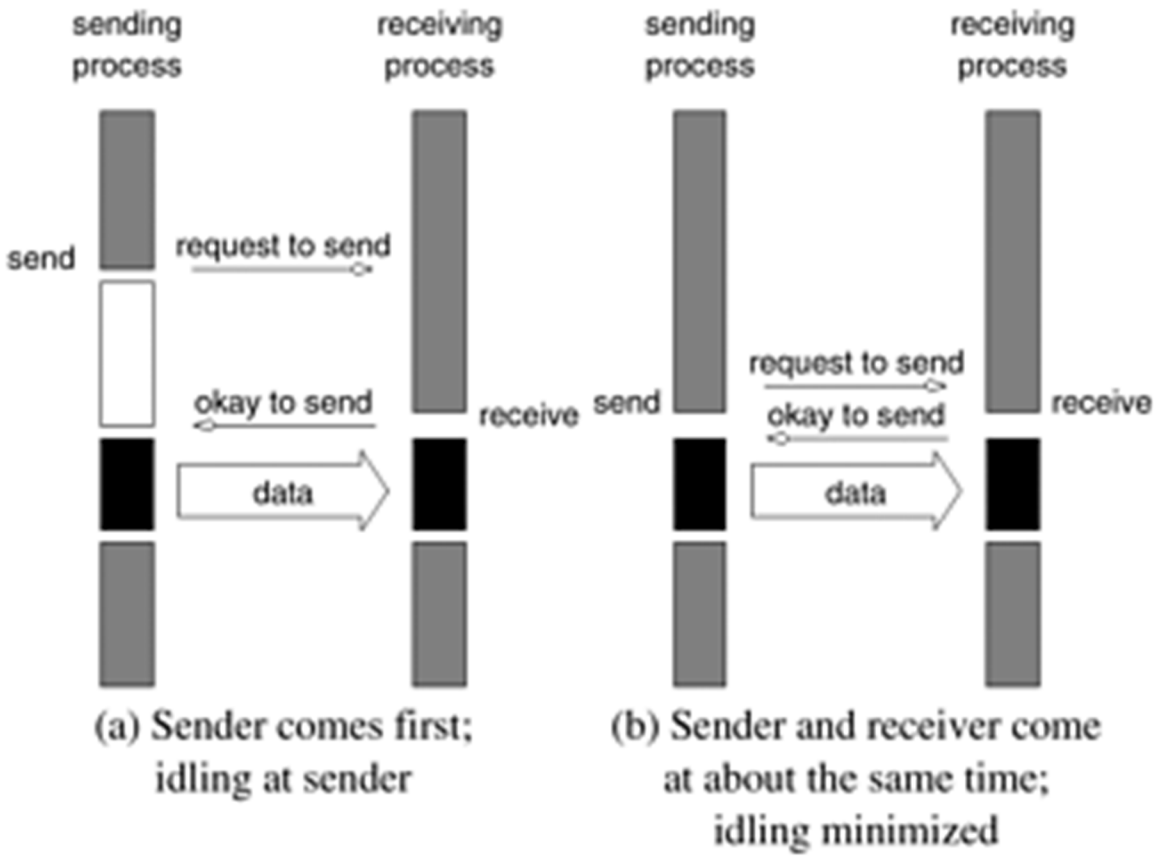
May lead to idling:



(a) Sender comes first; idling at sender

Blocking Non-Buffered Send / Receive

May lead to idling:

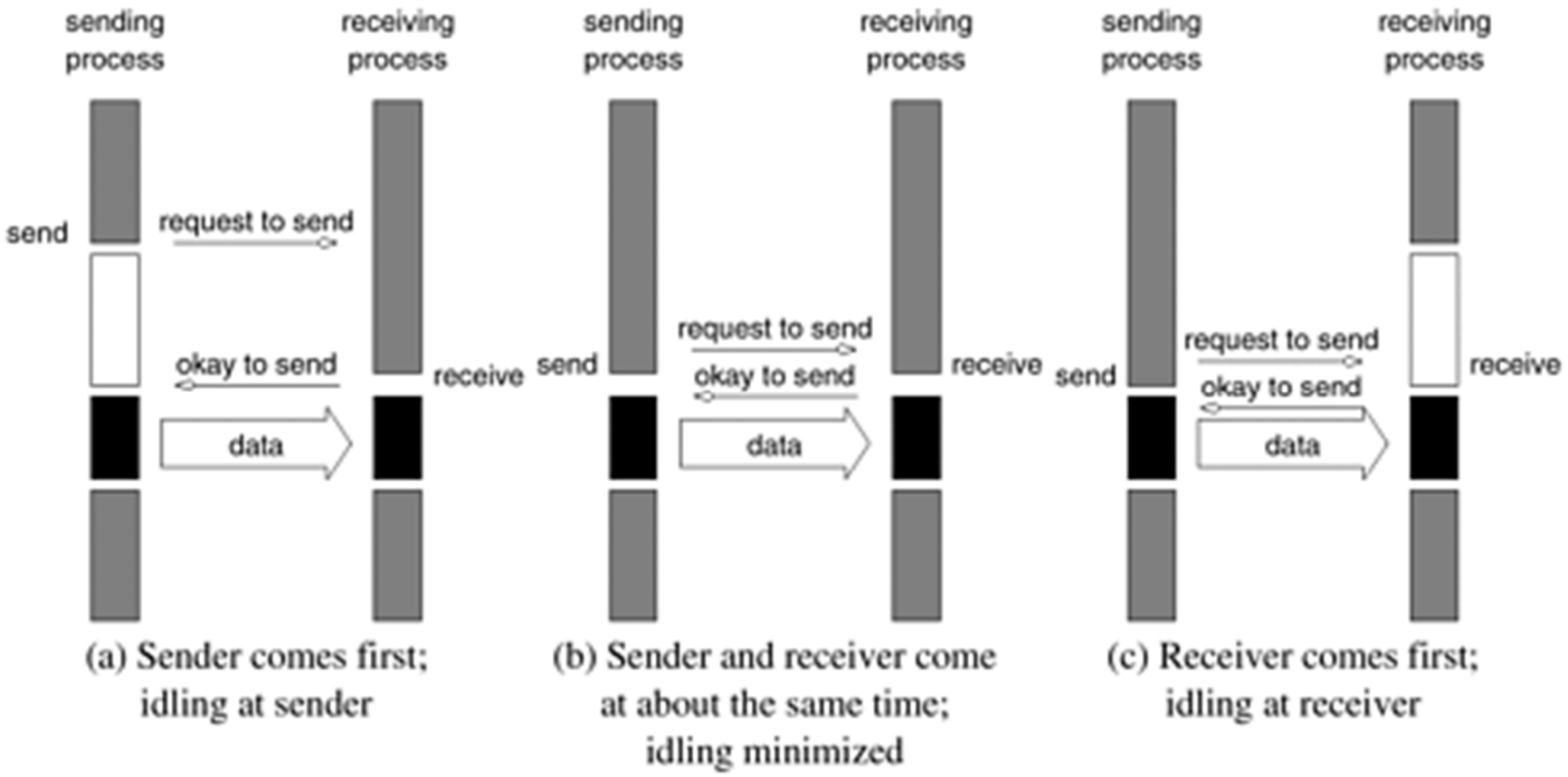


(a) Sender comes first; idling at sender

(b) Sender and receiver come at about the same time; idling minimized

Blocking Non-Buffered Send / Receive

May lead to idling:



(a) Sender comes first; idling at sender

(b) Sender and receiver come at about the same time; idling minimized

(c) Receiver comes first; idling at receiver

Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

Blocking Non-Buffered Send / Receive

May lead to deadlocks:

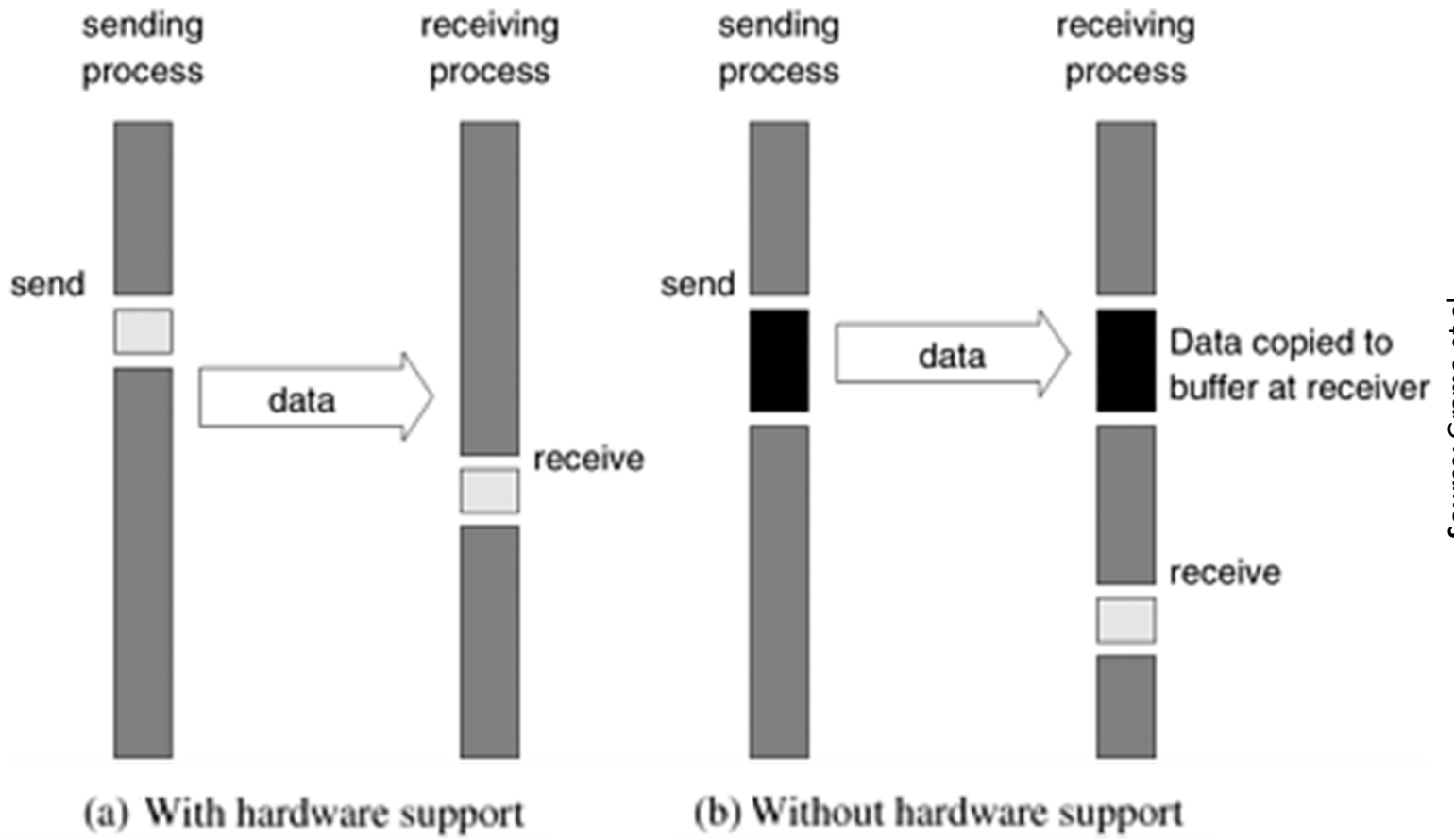
```
1          P0                                P1
2
3          send(&a, 1, 1);                   send(&a, 1, 0);
4          receive(&b, 1, 1);                receive(&b, 1, 0);
```

Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

- The send at P0 waits for the matching receive at P1
- The send at P1 waits for the matching receive at P0

Blocking Buffered Send / Receive

- Sending operation waits until data is copied into a pre-allocated communication buffer at the sending process
- Data is first copied into a buffer at the receiving process as well, from where data is copied to the target location by the receiver



Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

Blocking Buffered Send / Receive

Finite buffers lead to delays:

```
1          P0                                    P1
2
3          for (i = 0; i < 1000; i++) {          for (i = 0; i < 1000; i++) {
4            produce_data(&a);                     receive(&a, 1, 0);
5            send(&a, 1, 1);                       consume_data(&a);
6          }                                     }
```

Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

– What happens if the sender's buffer can only hold 10 items?

Blocking Buffered Send / Receive

May still lead to deadlocks:

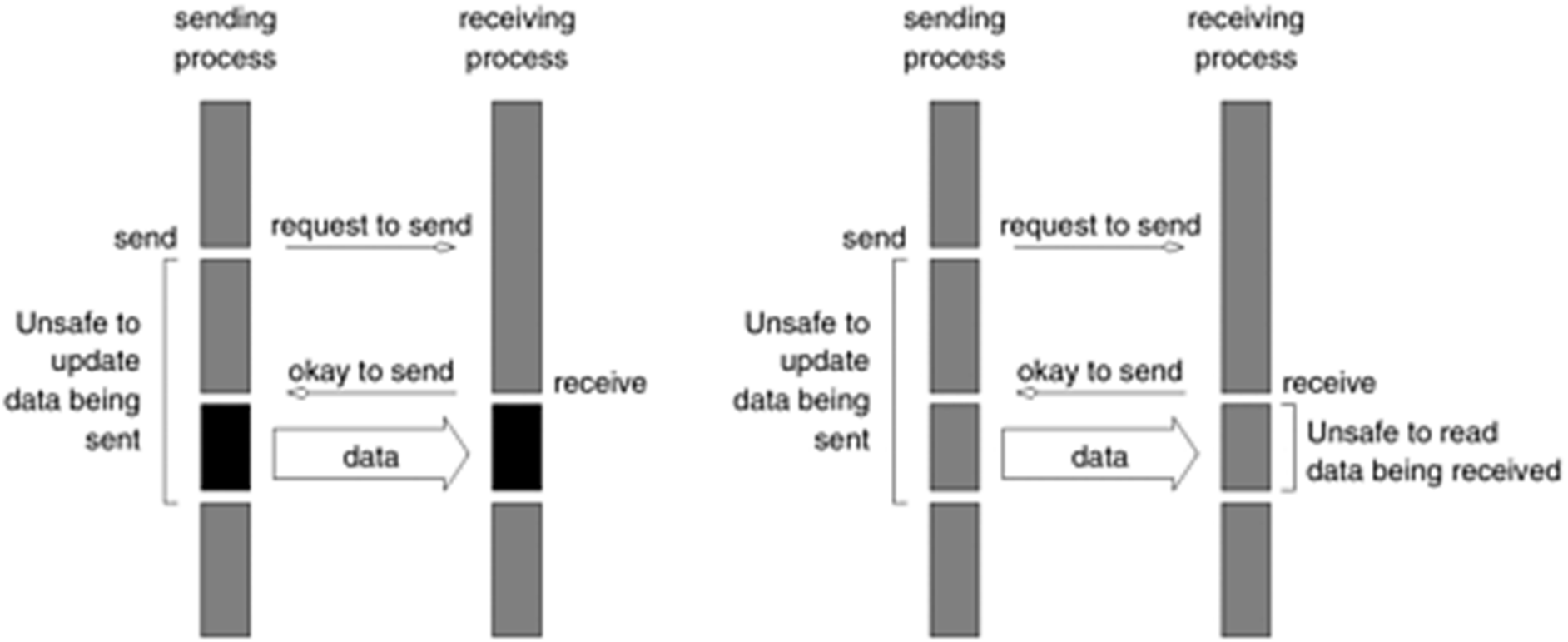
```
1          P0                                P1
2
3          receive(&a, 1, 1);                receive(&a, 1, 0);
4          send(&b, 1, 1);                   send(&b, 1, 0);
```

Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

– Blocks because the receive calls are always blocking in order to ensure consistency

Non-Blocking Non-Buffered Send / Receive

- Sending operation posts a pending message and returns
- When the corresponding receive is posted data transfer starts
- When data transfer is complete the *check-status* operation indicates that it is safe to touch the data

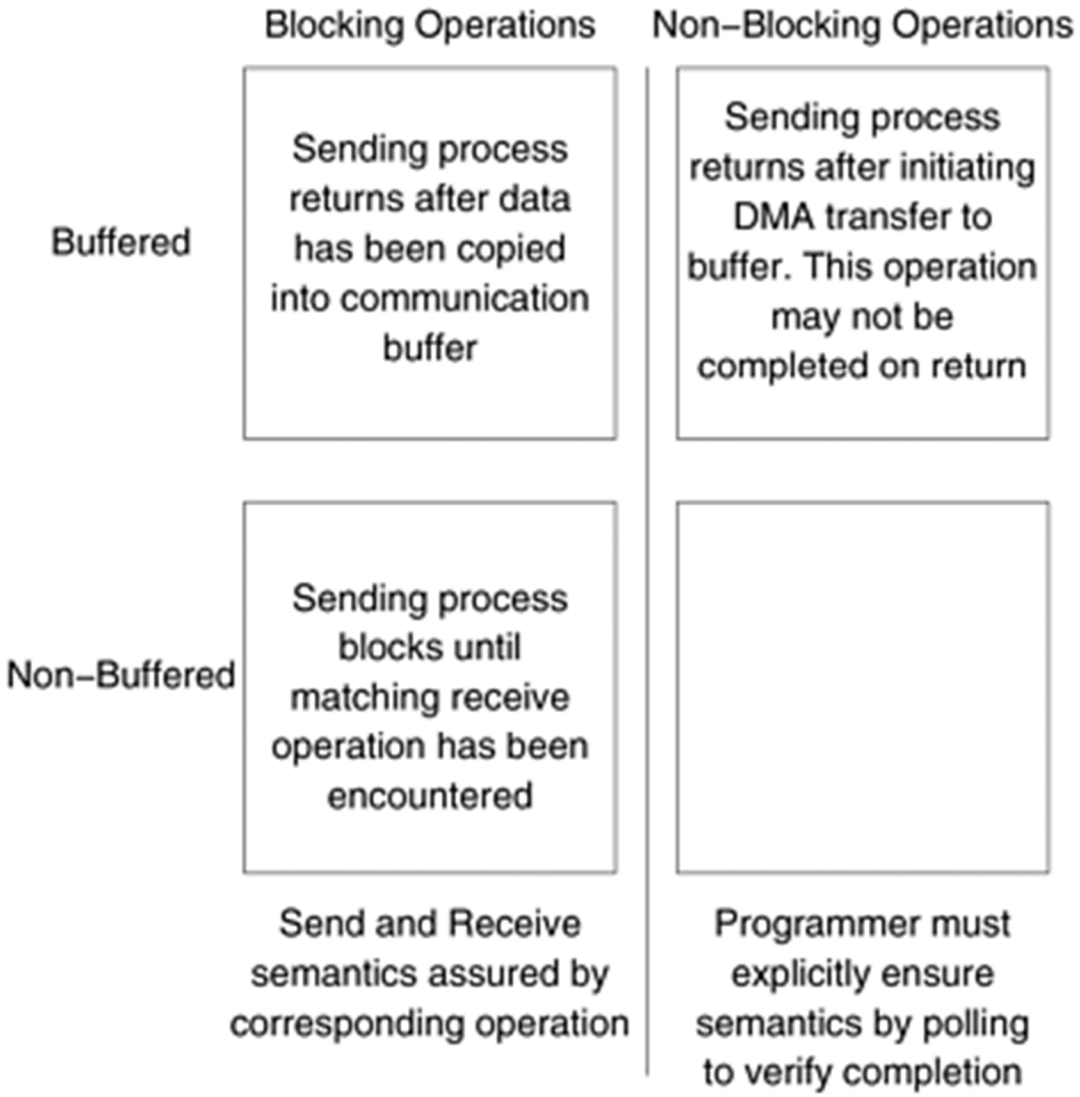


Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

Non-Blocking Buffered Send / Receive

- Sending operation initiates a DMA (Direct Memory Access) operation and returns immediately
- Data becomes safe as soon as the DMA operation completes
- The receiver initiates a transfer from sender's buffer to receiver's target location
- Reduces the time during which the data is unsafe to touch

Possible Protocols for Send & Receive Operations



Source: Grama et al.,
"Introduction to Parallel Computing",
2nd Edition

The Minimal Set of MPI Routines

- The MPI library contains over 125 routines
- But fully functional message-passing programs can be written using only the following 6 MPI routines

`MPI_Init`	Initializes MPI.
`MPI_Finalize`	Terminates MPI.
`MPI_Comm_size`	Determines the number of processes.
`MPI_Comm_rank`	Determines the label of the calling process.
`MPI_Send`	Sends a message.
`MPI_Recv`	Receives a message.

- All 6 functions return *MPI_SUCCESS* upon successful completion, otherwise return an implementation-defined error code
- All MPI routines, data-types and constants are prefixed by *MPI_*
- All of them are defined in *mpi.h* (for C/C++)

Starting and Terminating the MPI Library

```
1. #include < mpi.h >
2.
3. main( int argc, char *argv[ ] )
4. {
5.     MPI_Init( &argc, &argv );
6.     … … …                          // do some work
7.     MPI_Finalize( );
8. }
```

- Both *MPI_Init* and *MPI_Finalize* must be called by all processes
- Command line should be processed only after *MPI_Init*
- No MPI function may be called after *MPI_Finalize*

Communicators

- A *communicator* defines the scope of a communication operation
- Each process included in the communicator has a rank associated with the communicator
- By default, all processes are included in a communicator called *MPI_COMM_WORLD*, and each process is given a unique rank between 0 and $p - 1$, where p is the number of processes
- Additional communicator can be created for groups of processes
- To get the size of a communicator:

```
int MPI_Comm_size( MPI_Comm comm, int *size )
```

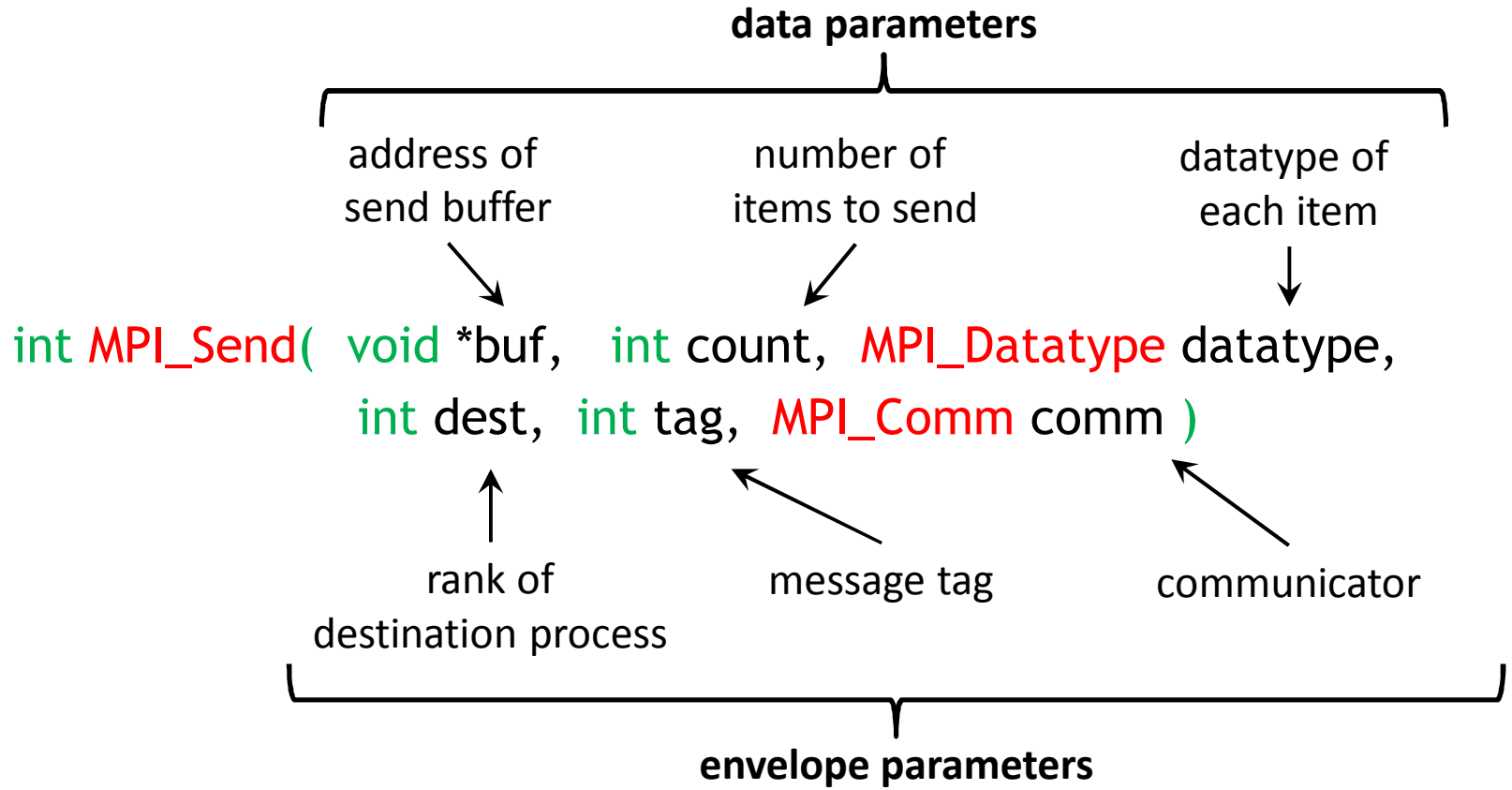
- To get the rank of a process associated with a communicator:

```
int MPI_Comm_rank( MPI_Comm comm, int *rank )
```

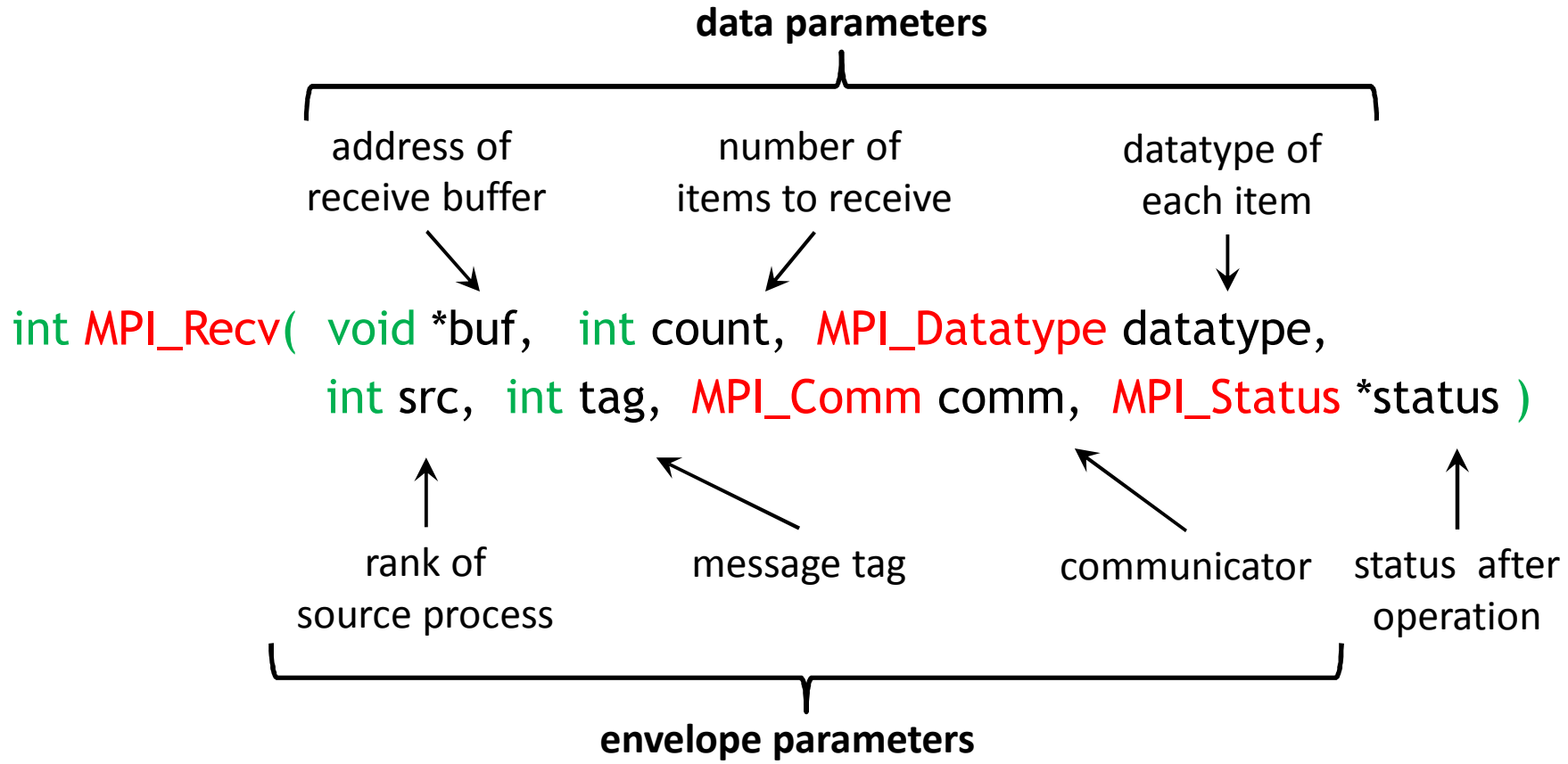
Communicators

```
1.  #include < mpi.h >
2.
3.  main( int argc, char *argv[ ] )
4.  {
5.      int p, myrank;
6.      MPI_Init( &argc, &argv );
7.      MPI_Comm_size( MPI_COMM_WORLD, &p );
8.      MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
9.      printf( "This is process %d out of %d!\n", p, myrank );
10.     MPI_Finalize( );
11. }
```

MPI Standard Blocking Send Format



MPI Standard Blocking Receive Format



MPI Datatypes

MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Blocking Send/Receive between Two Processes

```
1.  #include < mpi.h >
2.
3.  main( int argc, char *argv[ ] )
4.  {
5.      int myrank, v = 121;
6.      MPI_Status status;
7.      MPI_Init( &argc, &argv );
8.      MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
9.      if ( myrank == 0 ) {
10.         MPI_Send( &v, 1, MPI_INT,  1, MPI_ANY_TAG, MPI_COMM_WORLD );
11.         printf( "Process %d sent %d!\n", p, myrank, v );
12.     }  else if ( myrank == 1 ) {
13.         MPI_Recv( &v, 1, MPI_INT,  1, MPI_ANY_TAG, MPI_COMM_WORLD , &status );
14.         printf( "Process %d received %d!\n", p, myrank, v );
15.     }
16.     MPI_Finalize( );
17. }
```

Non-Blocking Send / Receive

```
int MPI_Isend( void *buf, int count, MPI_Datatype datatype,
               int dest, int tag, MPI_Comm comm, MPI_Request *req )

int MPI_Irecv( void *buf, int count, MPI_Datatype datatype,
               int src, int tag, MPI_Comm comm, MPI_Request *req )
```

The MPI_Request object is used as an argument to the following two functions to identify the operation whose status we want to query or to wait for its completion.

```
int MPI_Test( MPI_Request *req, int *flag, MPI_Status *status )
```

- Returns *flag = 1, if the operation associated with *req has completed, otherwise returns *flag = 0

```
int MPI_Wait( MPI_Request *req, MPI_Status *status )
```

- Waits until the operation associated with *req completes

Non-Blocking Send and Blocking Receive

```
1.  #include < mpi.h >
2.
3.  main( int argc, char *argv[ ] )
4.  {
5.      int myrank, v = 121;
6.      MPI_Status status;
7.      MPI_Request req;
8.      MPI_Init( &argc, &argv );
9.      MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
10.     if ( myrank == 0 ) {
11.         MPI_Isend( &v, 1, MPI_INT,  1, MPI_ANY_TAG, MPI_COMM_WORLD, &req );
12.         compute( );             /* but do not modify v */
13.         MPI_Wait( &req, &status );
14.     }  else if ( myrank == 1 ) MPI_Recv( &v, 1, MPI_INT,  1, MPI_ANY_TAG, MPI_COMM_WORLD , &status );
15.     MPI_Finalize( );
16. }
```

Non-Blocking Send/Receive

```
1.  #include < mpi.h >
2.  main( int argc, char *argv[ ] )
3.  {
4.      int myrank, v = 121;
5.      MPI_Status status;
6.      MPI_Request req;
7.      MPI_Init( &argc, &argv );
8.      MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
9.      if ( myrank == 0 ) {
10.         MPI_Isend( &v, 1, MPI_INT,  1, MPI_ANY_TAG, MPI_COMM_WORLD, &req );
11.         compute( );             /* but do not modify v */
12.         MPI_Wait( &req, &status );
13.     } else if ( myrank == 1 ) {
14.         MPI_Irecv( &v, 1, MPI_INT,  1, MPI_ANY_TAG, MPI_COMM_WORLD, &req );
15.         compute( );             /* but do not read or modify v */
16.         MPI_Wait( &req, &status );
17.     }
18.     MPI_Finalize( );
19. }
```

MPI Collective Communication & Computation Operations

Synchronization

- Barrier

Data Movement

- Broadcast
- Scatter
- Gather
- All-to-all

Global Computation

- Reduce
- Scan

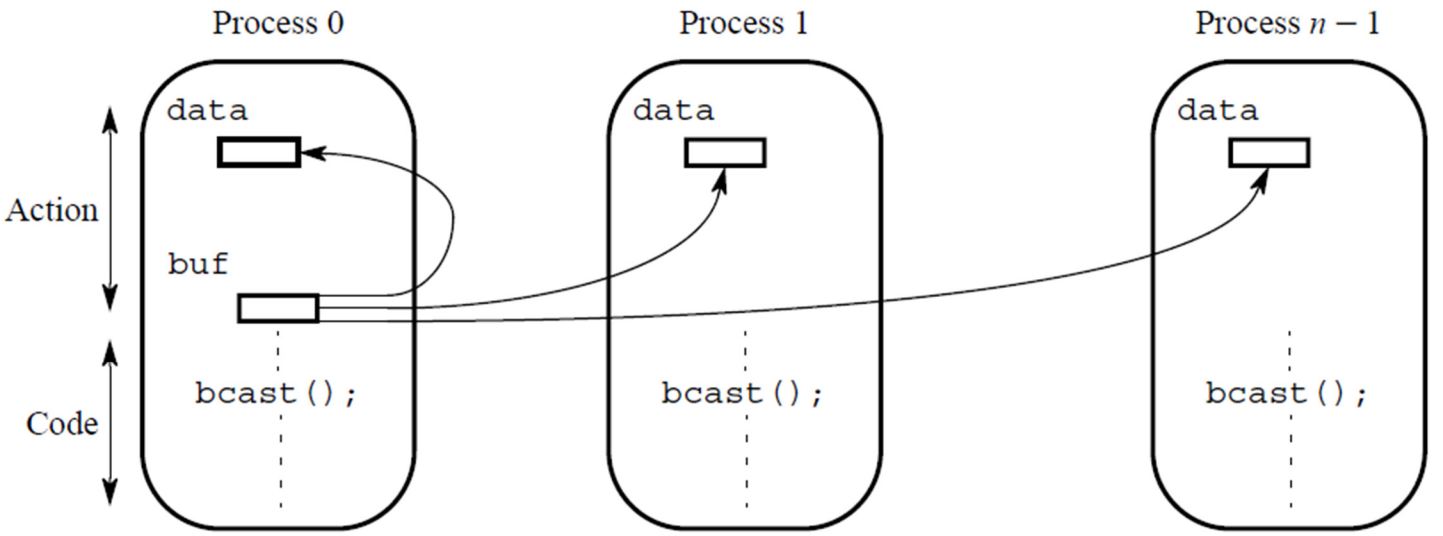
These routines must be called by all processes in the communication group

Barrier Synchronization

```
int MPI_Barrier( MPI_Comm comm )
```

Returns only after all processes in the communication group have called this function

Broadcast

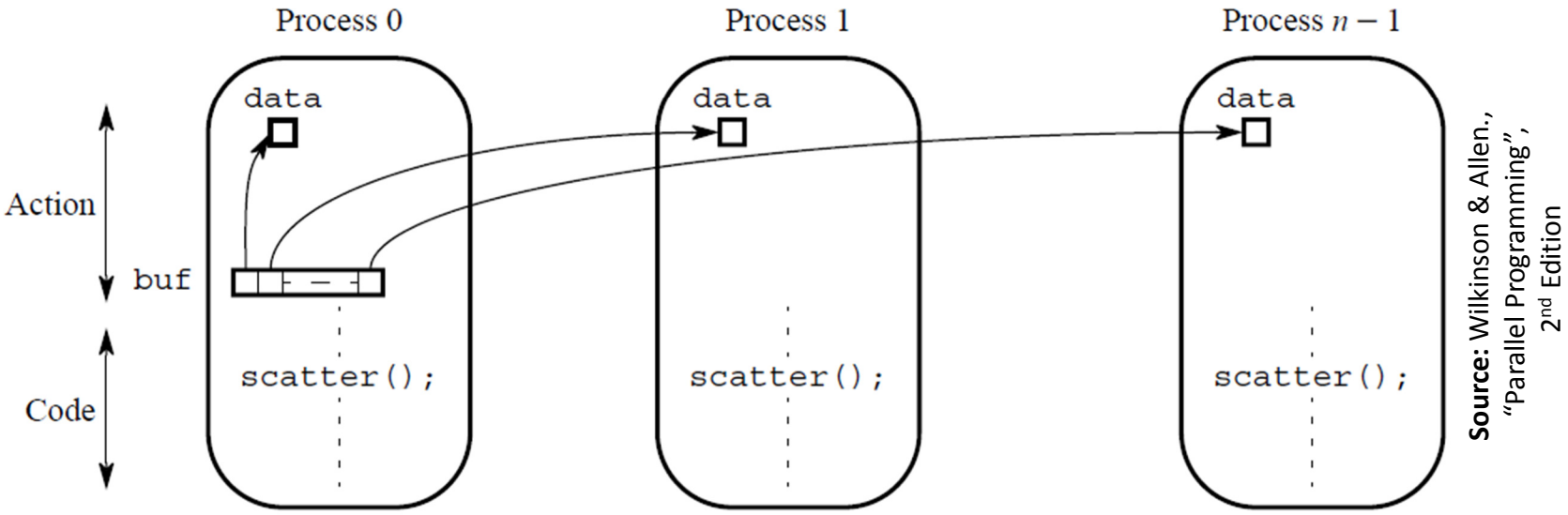


Source: Wilkinson & Allen., "Parallel Programming", 2nd Edition

```
int MPI_Bcast( void *buf,
               int count,
               MPI_Datatype datatype,
               int src,
               MPI_Comm comm )
```

Sends the data stored in the buffer *buf* of process *src* to all the other processes in the group

Scatter



Source: Wilkinson & Allen., "Parallel Programming", 2nd Edition

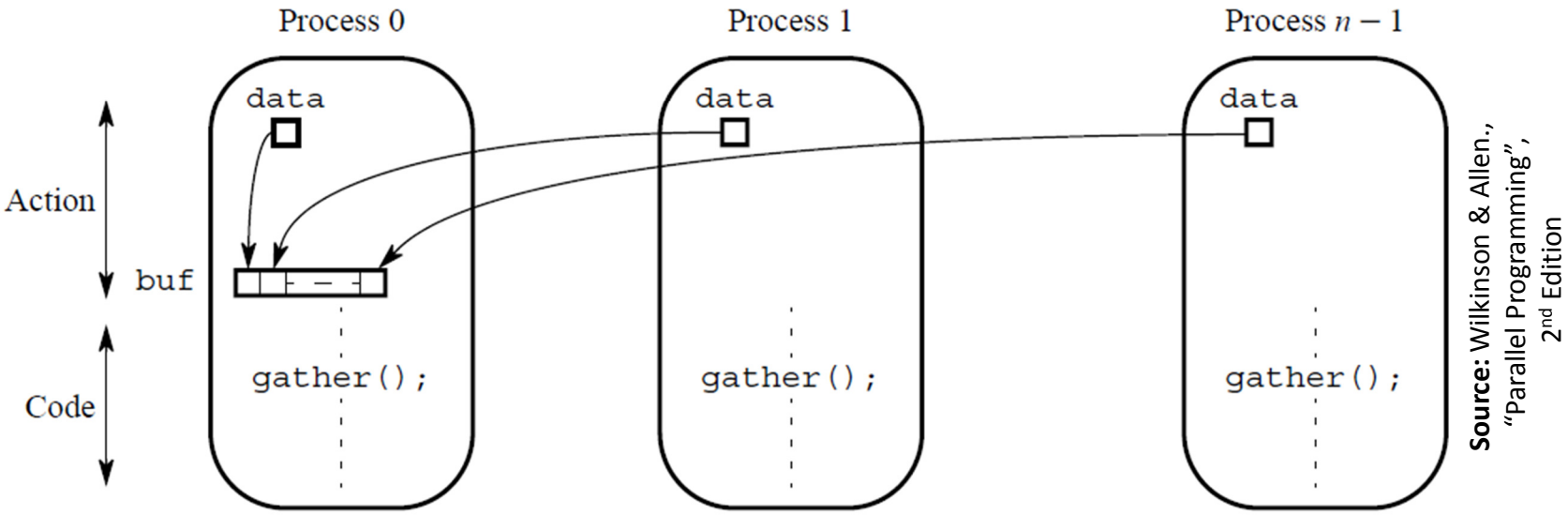
```
int MPI_Scatter( void *sendbuf,
                 int sendcount,
                 MPI_Datatype sendtype,
                 void *recvbuf,
                 int recvcount,
                 MPI_Datatype recvtype,
                 int src,
                 MPI_Comm comm )
```

The *src* process sends a different part of *sendbuf* to each process, including itself.

Process *i* receives *sendcount* contiguous elements starting from $i \times sendcount$.

The received data are stored in *recvbuf*.

Gather



Source: Wilkinson & Allen., "Parallel Programming", 2nd Edition

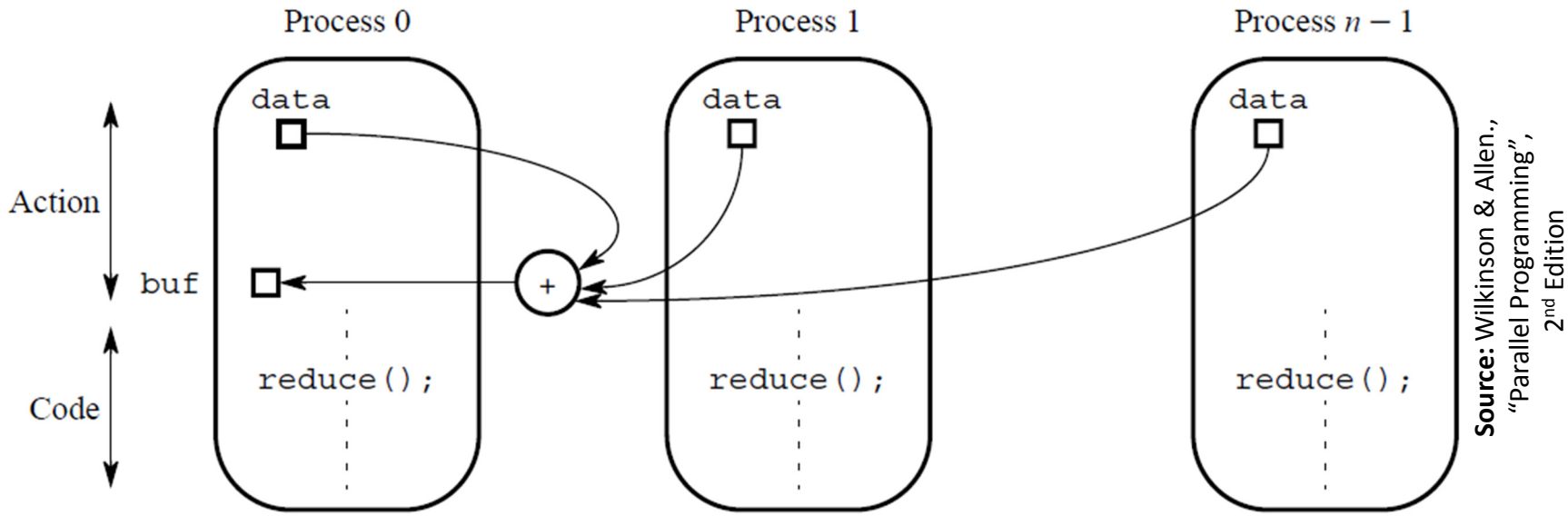
```
int MPI_Gather( void *sendbuf,
                int sendcount,
                MPI_Datatype sendtype,
                void *recvbuf,
                int recvcount,
                MPI_Datatype recvtype,
                int dest,
                MPI_Comm comm )
```

The opposite of scatter.

Every process, including *dest* sends data stored in *sendbuf* to *dest*.

Data from process *i* occupy *sendcount* contiguous locations of *recvbuf* starting from $i \times sendcount$.

Reduce

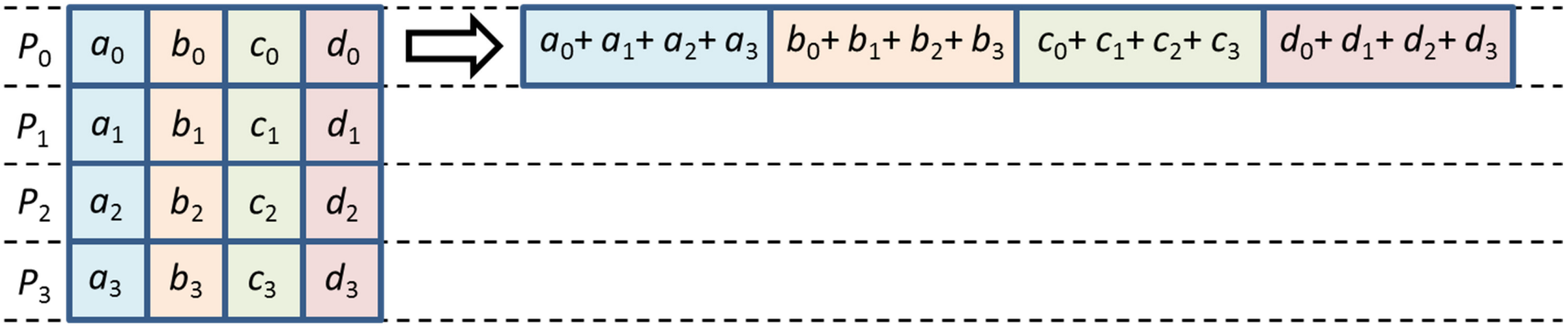



```
int MPI_Reduce( void *sendbuf,
                void *recvbuf,
                int count,
                MPI_Datatype datatype,
                MPI_Op op,
                int dest,
                MPI_Comm comm )
```

Combines the elements stored in *sendbuf* of each process using the operation *op*, and stores the combined values in *recvbuf* of the process with rank *dest*.

Reduce

MPI_Reduce(vals, sums, 4, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)



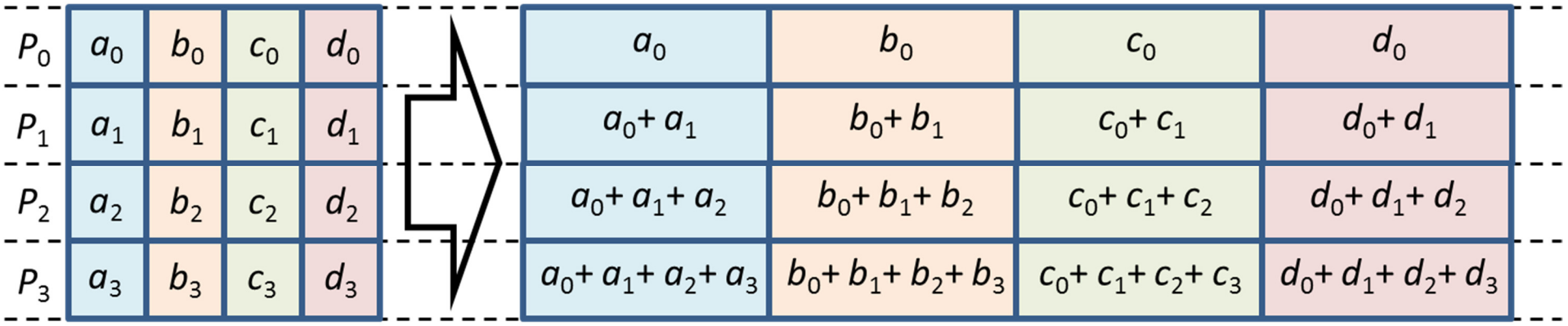
Predefined Reduction Operations

Operation	Meaning	Datatypes
MPI_MAX	Maximum	C integers and floating point
MPI_MIN	Minimum	C integers and floating point
MPI_SUM	Sum	C integers and floating point
MPI_PROD	Product	C integers and floating point
MPI_LAND	Logical AND	C integers
MPI_BAND	Bit-wise AND	C integers and byte
MPI_LOR	Logical OR	C integers
MPI_BOR	Bit-wise OR	C integers and byte
MPI_LXOR	Logical XOR	C integers
MPI_BXOR	Bit-wise XOR	C integers and byte
MPI_MAXLOC	max-min value-location	Data-pairs
MPI_MINLOC	min-min value-location	Data-pairs

Scan / Prefix

```
int MPI_Scan( void *sendbuf,
              void *recvbuf,
              int count,
              MPI_Datatype datatype,
              MPI_Op op,
              MPI_Comm comm )
```

Performs a prefix reduction of the data stored in *sendbuf* at each process and returns the results in *recvbuf* of the process with rank *dest*.




```
MPI_Scan( vals, sums, 4, MPI_INT, MPI_SUM, MPI_COMM_WORLD )
```

Topologies and Embedding

Instructor
Dr B Krishna Priya

Topologies and Embedding

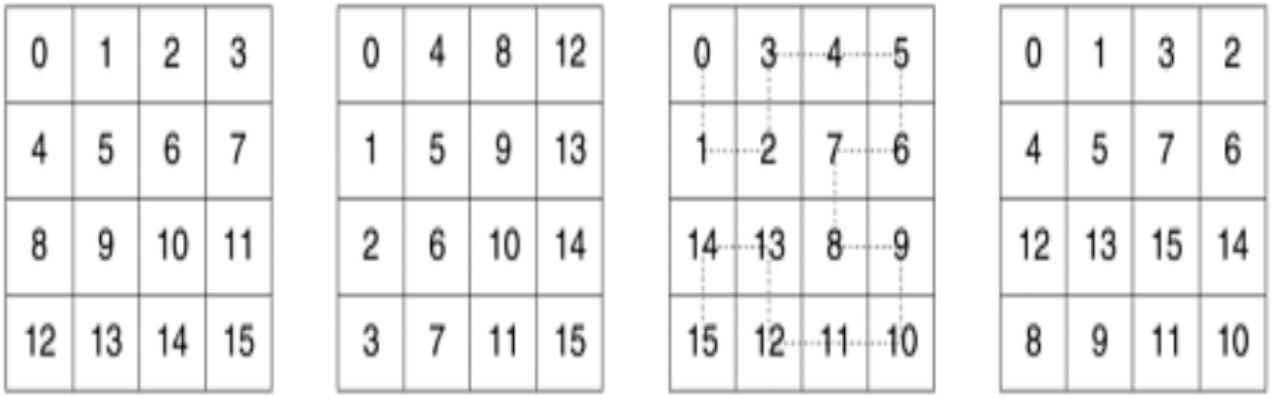
- Processes as being arranged in a one-dimensional topology and uses a linear ordering to number the processes.
- In parallel programs:
- processes are naturally arranged in higher-dimensional topologies (e.g., two- or three-dimensional)
- computation and the set of interacting processes are naturally identified by their coordinates in that topology.

Contd..

- In a parallel program in which the processes are arranged in a two-dimensional topology, process (i , j) may need to send message to (or receive message from) process (k , l).
- An MPI process with rank rank corresponds to process (row , col) in the grid such that row = rank/4 and col = rank%4

Contd..

Figure 6.5. Different ways to map a set of processes to a two-dimensional grid. (a) and (b) show a row- and column-wise mapping of these processes, (c) shows a mapping that follows a space-filling curve (dotted line), and (d) shows a mapping in which neighboring processes are directly connected in a hypercube.



(a) Row-major mapping (b) Column-major mapping (c) Space-filling curve mapping (d) Hypercube mapping

Contd..

- MPI provides a set of routines that allows the programmer to arrange the processes in different topologies without having to explicitly specify how these processes are mapped onto the processors.
- It is up to the MPI library to find the most appropriate mapping that reduces the cost of sending and receiving messages.

Creating and Using Cartesian Topologies

- Graphs of processes can be used to specify any desired topology. However, most commonly used topologies in message-passing programs are one-, two-, or higher-dimensional grids, that are also referred to as Cartesian topologies

Contd..

- MPI's function for describing Cartesian topologies is called MPI_Cart_create .
- Its calling sequence is as follows.

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims,
int *dims, int *periods, int reorder, MPI_Comm
*comm_cart)
```

- This function takes the group of processes that belong to the communicator comm_old and creates a virtual process topology.
- The topology information is attached to a new communicator comm_cart that is created by MPI_Cart_create .

Contd..

- The shape and properties of the topology are specified by the arguments ndims , dims , and periods.
- ndims: specifies the number of dimensions of the topology.
- dims: specify the size along each dimension of the topology.
- periods:specify whether or not the topology has wraparound connections.
- periods[i] is true (non-zero in C), then the topology has wraparound connections along dimension i , otherwise it does not.

Contd.

- reorder:if the processes in the new group (i.e., communicator) are to be reordered or not.
- If reorder is false, then the rank of each process in the new group is identical to its rank in the old group.
- Total number of processes specified in the dims array is smaller than the number of processes in the communicator specified by comm_old , then some processes will not be part of the Cartesian topology. For this set of processes, the value of comm_cart will be set to MPI_COMM_NULL (an MPI defined constant).
- If it is greater, it will leads to error.

Process naming

- MPI provides two functions, MPI_Cart_rank and MPI_Cart_coord , for performing coordinate-to-rank and rank-to-coordinate translations, respectively.
- The calling sequences of these routines are the following:
- int MPI_Cart_rank(MPI_Comm comm_cart, int *coords, int *rank)
- int MPI_Cart_coord(MPI_Comm comm_cart, int rank, int maxdims, int *coords)
- The MPI_Cart_rank takes the coordinates of the process as argument in the coords array and returns its rank in rank .

- The MPI_Cart_coords takes the rank of the process rank and returns its Cartesian coordinates in the array coords , of length maxdims.
- Note:maxdims should be at least as large as the number of dimensions in the Cartesian topology specified by the communicator comm_cart .

- The communication performed among processes in a Cartesian topology is that of shifting data along a dimension of the topology.
- int MPI_Cart_shift(MPI_Comm comm_cart, int dir, int s_step, int *rank_source, int *rank_dest)
- dir: direction of the shift
- s_step: size of the shift step
- The computed ranks are returned in rank_source and rank_dest.

Programming Using the Message Passing Paradigm

Chieh-Sen (Jason) Huang

Department of Applied Mathematics

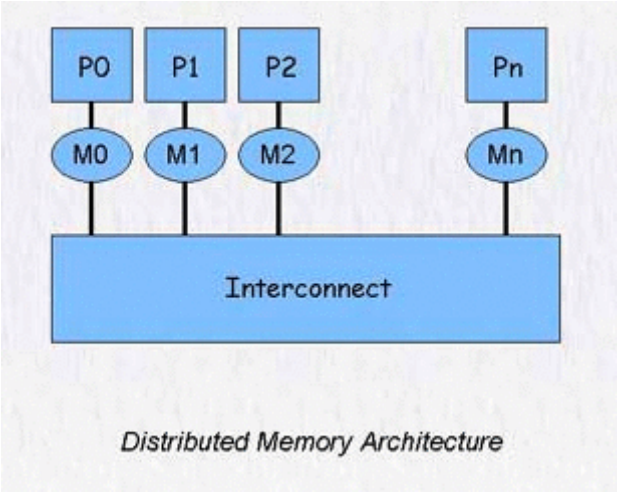
National Sun Yat-sen University

Thank Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar for providing slides.

Topic Overview

- Principles of Message-Passing Programming
- The Building Blocks: Send and Receive Operations
- MPI: the Message Passing Interface

Principles of Message-Passing Programming



- The logical view of a machine supporting the message-passing paradigm consists of p processes, each with its own exclusive address space.
- Each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed.
- All interactions (read-only or read/write) require cooperation of two processes – the process that has the data and the process that wants to access the data.
- These two constraints, while onerous, make underlying costs very explicit to the programmer.

Principles of Message-Passing Programming

- Message-passing programs are often written using the *asynchronous* or *loosely synchronous* paradigms.

- In the asynchronous paradigm, all concurrent tasks execute asynchronously.

- In the loosely synchronous model, tasks or subsets of tasks synchronize to perform interactions. Between these interactions, tasks execute completely asynchronously.

- Most message-passing programs are written using the *single program multiple data* (SPMD) model.

The Building Blocks: Send and Receive Operations

- The prototypes of these operations are as follows:

```
send(void *sendbuf, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

- Consider the following code segments:

```
P0                          P1

a = 100;                    receive(&a, 1, 0)
send(&a, 1, 1);             printf("%d\n", a);
a = 0;
```

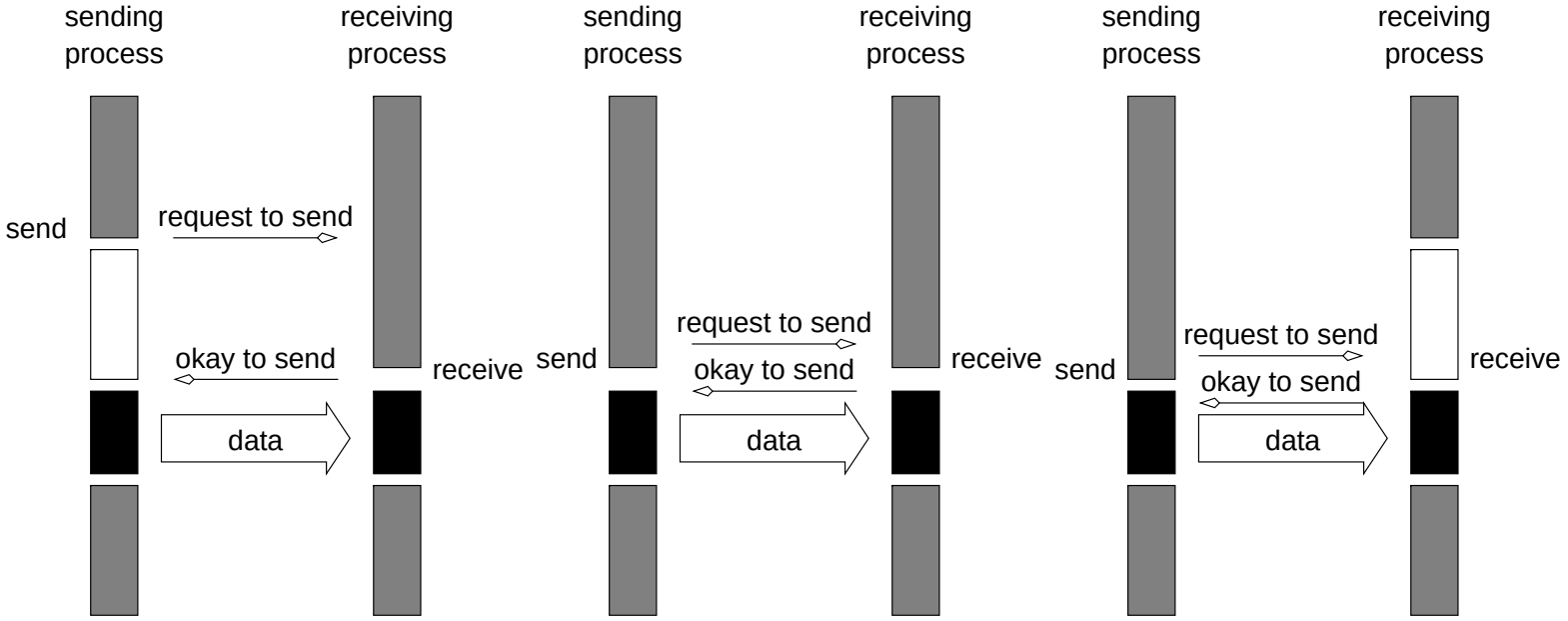
- The semantics of the send operation require that the value received by process P1 must be 100 as opposed to 0.

- This motivates the design of the send and receive protocols.

Non-Buffered Blocking Message Passing Operations

- A simple method for forcing send/receive semantics is for the send operation to return only when it is safe to do so.
- In the non-buffered blocking send, the operation does not return until the matching receive has been encountered at the receiving process.
- Idling and deadlocks are major issues with non-buffered blocking sends.
- In buffered blocking sends, the sender simply copies the data into the designated buffer and returns after the copy operation has been completed. The data is copied at a buffer at the receiving end as well.
- Buffering alleviates idling at the expense of copying overheads.

Non-Buffered Blocking Message Passing Operations



(a) Sender comes first; idling at sender

(b) Sender and receiver come at about the same time; idling minimized

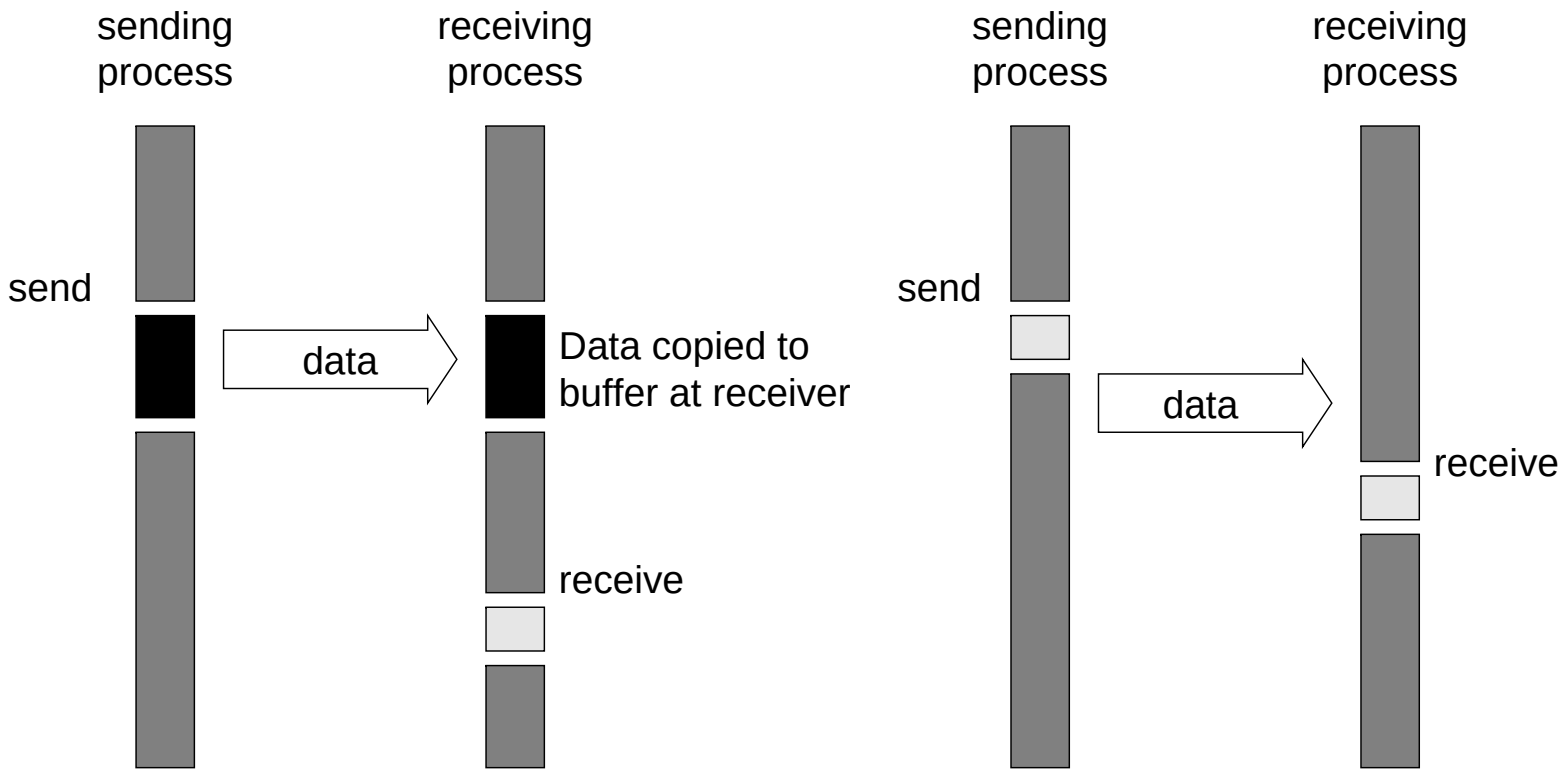
(c) Receiver comes first; idling at receiver

Handshake for a blocking non-buffered send/receive operation. It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.

Buffered Blocking Message Passing Operations

- A simple solution to the idling and deadlocking problem outlined above is to rely on buffers at the sending and receiving ends.
- The sender simply copies the data into the designated buffer and returns after the copy operation has been completed.
- The data must be buffered at the receiving end as well.
- Buffering trades off idling overhead for buffer copying overhead.

Buffered Blocking Message Passing Operations



Blocking buffered transfer protocols: (a) in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end; and (b) in the presence of communication hardware with buffers at send and receive ends.

Buffered Blocking Message Passing Operations

Bounded buffer sizes can have significant impact on performance.

```
P0

for (i = 0; i < 1000; i++) {
  produce_data(&a);
  send(&a, 1, 1);
}
```

```
P1

for (i = 0; i < 1000; i++) {
  receive(&a, 1, 0);
  consume_data(&a);
}
```

What if consumer was much slower than producer?

Buffered Blocking Message Passing Operations

Deadlocks are still possible with buffering since receive operations block.

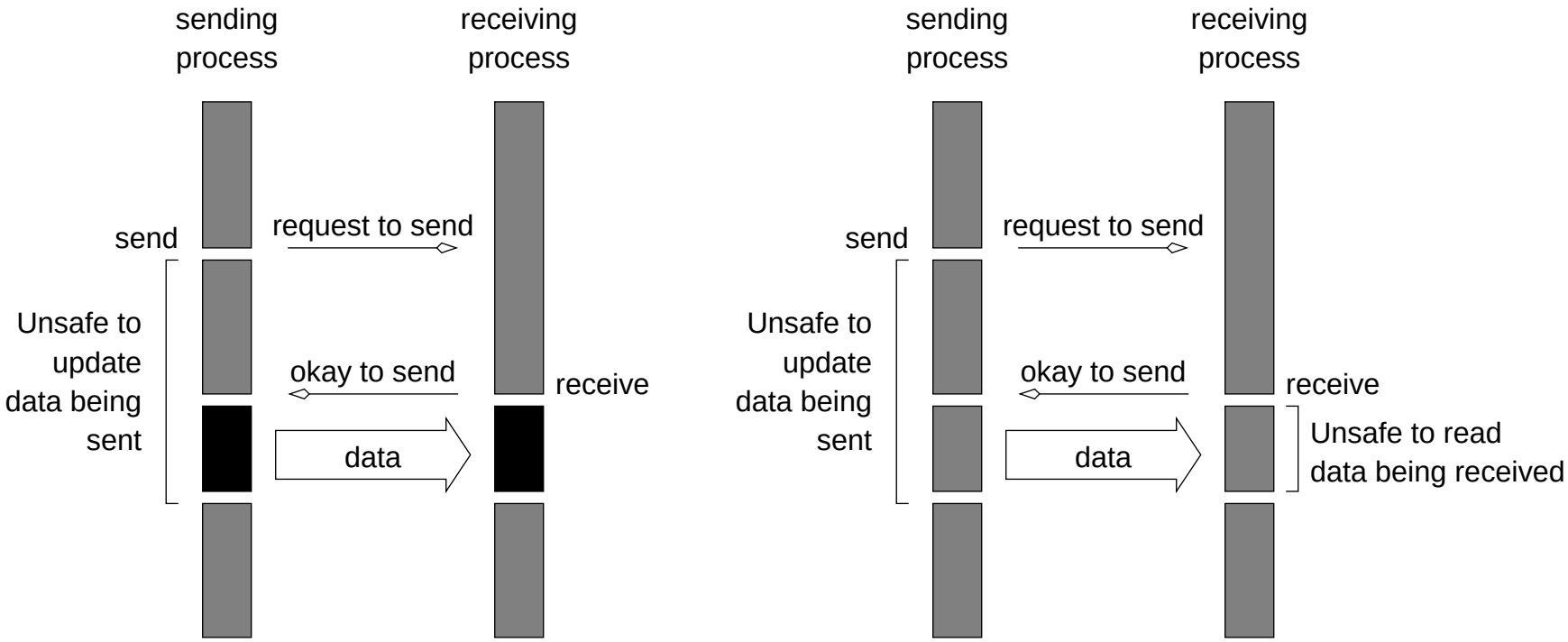
```
P0                              P1

receive(&a, 1, 1);              receive(&a, 1, 0);
send(&b, 1, 1);                 send(&b, 1, 0);
```

Non-Blocking Message Passing Operations

- The programmer must ensure semantics of the send and receive.
- This class of non-blocking protocols returns from the send or receive operation before it is semantically safe to do so.
- Non-blocking operations are generally accompanied by a `check-status` operation.
- When used correctly, these primitives are capable of overlapping communication overheads with useful computations.
- Message passing libraries typically provide both blocking and non-blocking primitives.

Non-Blocking Message Passing Operations

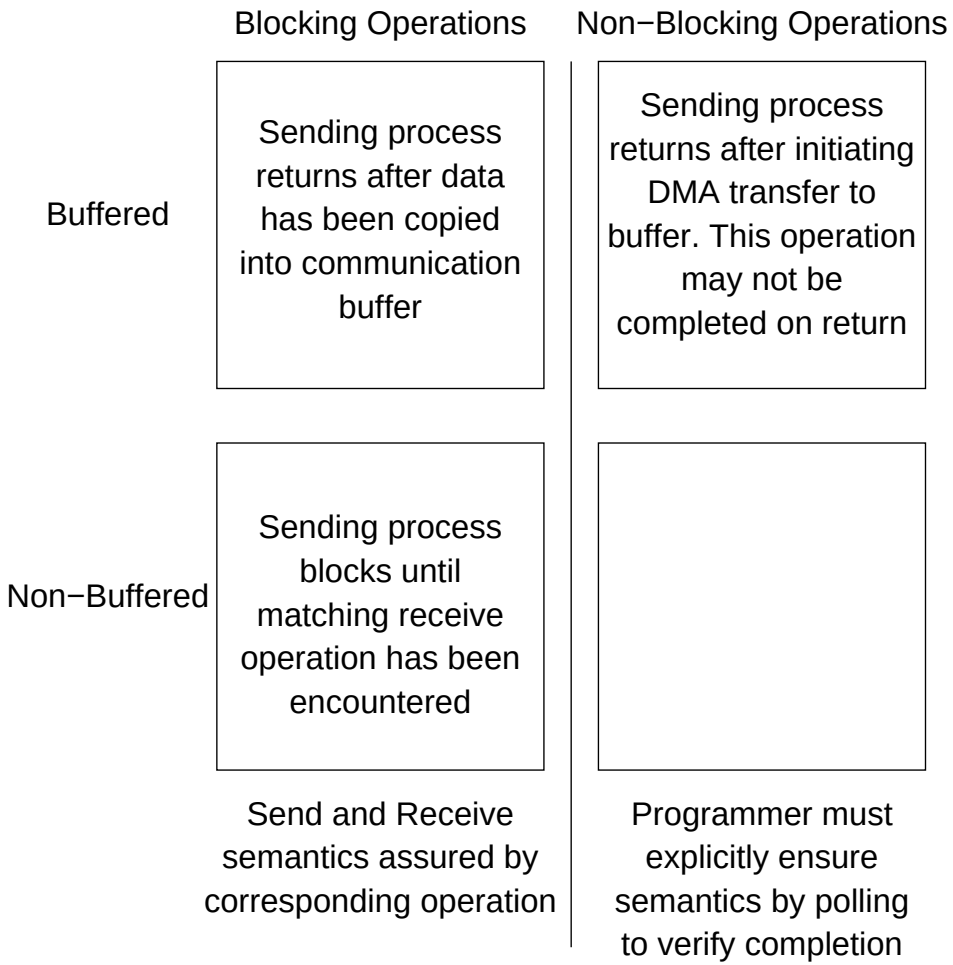


(a) Without hardware support

(b) With hardware support

Non-blocking non-buffered send and receive operations (a) in absence of communication hardware; (b) in presence of communication hardware.

Send and Receive Protocols



Space of possible protocols for send and receive operations.

MPI: the Message Passing Interface

- MPI defines a standard library for message-passing that can be used to develop portable message-passing programs using either C or Fortran.

- The MPI standard defines both the syntax as well as the semantics of a core set of library routines.

- Vendor implementations of MPI are available on almost all commercial parallel computers.

- It is possible to write fully-functional message-passing programs by using only the six routines.

MPI: the Message Passing Interface

The minimal set of MPI routines.

`MPI_Init`	Initializes MPI.
`MPI_Finalize`	Terminates MPI.
`MPI_Comm_size`	Determines the number of processes.
`MPI_Comm_rank`	Determines the label of the calling process.
`MPI_Send`	Sends a message.
`MPI_Recv`	Receives a message.

Starting and Terminating the MPI Library

- `MPI_Init` is called prior to any calls to other MPI routines. Its purpose is to initialize the MPI environment.

- `MPI_Finalize` is called at the end of the computation, and it performs various clean-up tasks to terminate the MPI environment.

- The prototypes of these two functions are:

```
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize()
```

- `MPI_Init` also strips off any MPI related command-line arguments.

- All MPI routines, data-types, and constants are prefixed by "`MPI_`". The return code for successful completion is `MPI_SUCCESS`.

Communicators

- A communicator defines a *communication domain* – a set of processes that are allowed to communicate with each other.
- Information about communication domains is stored in variables of type `MPI_Comm`.
- Communicators are used as arguments to all message transfer MPI routines.
- A process can belong to many different (possibly overlapping) communication domains.
- MPI defines a default communicator called `MPI_COMM_WORLD` which includes all the processes.

Querying Information

- The `MPI_Comm_size` and `MPI_Comm_rank` functions are used to determine the number of processes and the label of the calling process, respectively.

- The calling sequences of these routines are as follows:

```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

- The rank of a process is an integer that ranges from zero up to the size of the communicator minus one.

Our First MPI Program

```
#include <mpi.h>

main(int argc, char *argv[])
{
  int npes, myrank;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &npes);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  printf("From process %d out of %d, Hello World!\n",
         myrank, npes);
  MPI_Finalize();
}
```

Sending and Receiving Messages

- The basic functions for sending and receiving messages in MPI are the `MPI_Send` and `MPI_Recv`, respectively.

- The calling sequences of these routines are as follows:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
        int dest, int tag, MPI_Comm comm)
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
        int source, int tag, MPI_Comm comm, MPI_Status *status)
```

- MPI provides equivalent datatypes for all C datatypes. This is done for portability reasons.

- The datatype `MPI_BYTE` corresponds to a byte (8 bits) and `MPI_PACKED` corresponds to a collection of data items that has been created by packing non-contiguous data.

- The message-`tag` can take values ranging from zero up to the MPI defined constant `MPI_TAG_UB`.

MPI Datatypes

MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Sending and Receiving Messages

- MPI allows specification of wildcard arguments for both `source` and `tag`.
- If `source` is set to `MPI_ANY_SOURCE`, then any process of the communication domain can be the source of the message.
- If `tag` is set to `MPI_ANY_TAG`, then messages with any tag are accepted.
- On the receive side, the message must be of length equal to or less than the length field specified.

Sending and Receiving Messages

- On the receiving end, the `status` variable can be used to get information about the `MPI_Recv` operation.

- The corresponding data structure contains:

```
typedef struct MPI_Status {
  int MPI_SOURCE;
  int MPI_TAG;
  int MPI_ERROR;
};
```

- The `MPI_Get_count` function returns the precise count of data items received.

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype,
        int *count)
```

Avoiding Deadlocks

Consider:

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
  MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
  MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
  MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
  MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
...
```

If `MPI_Send` is blocking, there is a deadlock.
IBM's implementation doesn't seem to be a blocking operation, not what was written in the documentation, though.

Avoiding Deadlocks

Consider the following piece of code, in which process i sends a message to process $i + 1$ (modulo the number of processes) and receives a message from process $i - 1$ (module the number of processes).

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
...
```

Once again, we have a deadlock if `MPI_Send` is blocking.

Avoiding Deadlocks

We can break the circular wait to avoid deadlocks as follows:

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank%2 == 1) {
  MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
  MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
}
else {
  MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
  MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
}
...
```

Sending and Receiving Messages Simultaneously

To exchange messages, MPI provides the following function:

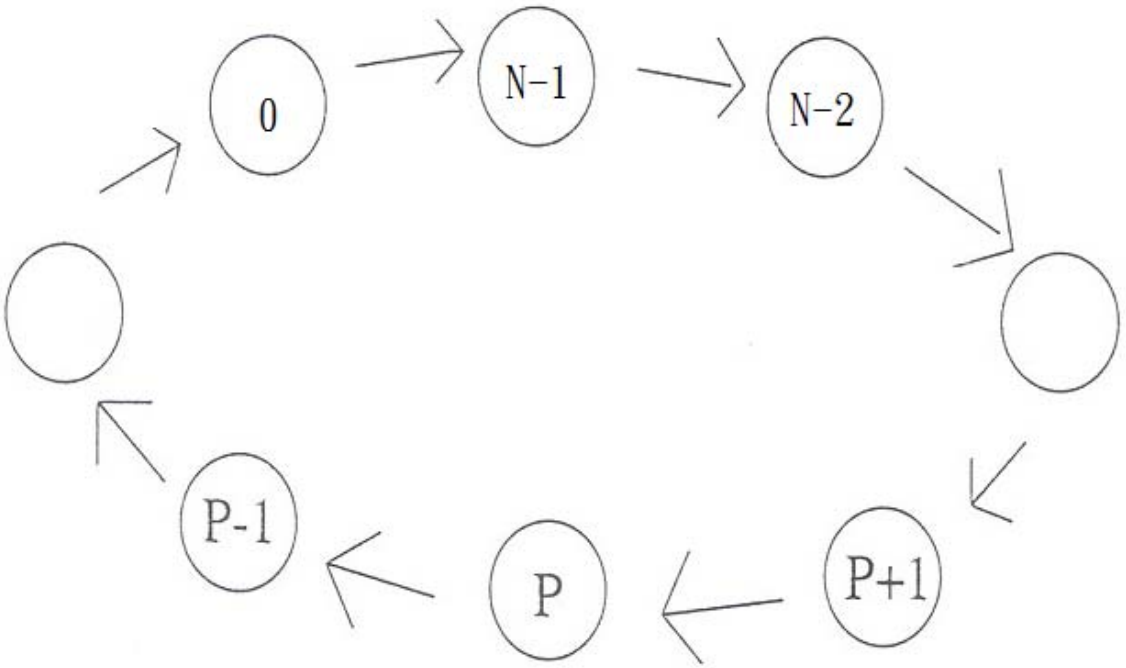
```
int MPI_Sendrecv(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, int dest, int sendtag,
        void *recvbuf, int recvcount, MPI_Datatype recvdatatype,
        int source, int recvtag, MPI_Comm comm,
        MPI_Status *status)
```

The arguments include arguments to the send and receive functions. If we wish to use the same buffer for both send and receive, we can use:

```
int MPI_Sendrecv_replace(void *buf, int count,
        MPI_Datatype datatype, int dest, int sendtag,
        int source, int recvtag, MPI_Comm comm,
        MPI_Status *status)
```

Output in a Reserve Order of Myrank

Can we output an array in a reserve order of myrank?



Output in a Reserve Order of Myrank

```
a[0] = myrank;
token = 0;

if (myrank == npes-1){
  printf("From processor: %d out of %d a=%d \n",myrank,npes,a[0]);
  token = 1;
  for(i=1;i<1000000;i++){}
  MPI_Send(&token,1,MPI_INT,myrank-1,1,MPI_COMM_WORLD);
  MPI_Recv(&token,1,MPI_INT,(myrank+1+npes)%npes,1,MPI_COMM_WORLD,
           &status);

}
else {
  MPI_Recv(&token,1,MPI_INT,(myrank+1+npes)%npes,1,MPI_COMM_WORLD,
           &status);
  printf("From processor: %d out of %d a=%d \n",myrank,npes,a[0]);

  for(i=1;i<1000000;i++){}
  token = 1;
  MPI_Send(&token,1,MPI_INT,(myrank-1+npes)%npes,1,MPI_COMM_WORLD);
}
```

Overlapping Communication with Computation

Instructor

Dr B. Krishna Priya

Non-Blocking Communication Operations

- MPI provides a pair of functions for performing non-blocking send and receive operations. These functions are MPI_Isend and MPI_Irecv.
- MPI_Isend starts a send operation but does not complete, that is, it returns before the data is copied out of the buffer.
- MPI_Irecv starts a receive operation but returns before the data has been received and copied into the buffer.
- With the support of appropriate hardware, the transmission and reception of messages can proceed concurrently with the computations performed by the program upon the return.

- The calling sequences of MPI_Isend and MPI_Irecv are the following:
- int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
- int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

- a process that has started a non-blocking send or receive operation must make sure that this operation has completed before it proceeds with its computations.
- This is because a process that has started a non-blocking send operation may want to overwrite the buffer that stores the data that are being sent, or a process that has started a non-blocking receive operation may want to use the data it requested.

- MPI provides a pair of functions MPI_Test and MPI_Wait.
- int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
- int MPI_Wait(MPI_Request *request, MPI_Status *status)
- The request objectin MPI_Isend and MPI_Irecv is used as an argument in the MPI_Test and MPI_Wait.

- MPI_Test tests whether or not the non-blocking send or receive operation identified by its request has finished.
- It returns flag = {true} (non-zero value in C) if it completed, otherwise it returns {false} (a zero value in C).
- non-blocking operation has finished, the request object pointed to by request is deallocated and request is set to MPI_REQUEST_NULL.

- The MPI_Wait function blocks until the non-blocking operation identified by request completes.
- In that case it deal-locates the request object, sets it to MPI_REQUEST_NULL, and returns information about the completed operation in the status object.

- The programmer wants to explicitly deallocate a request object, MPI provides the following function.
- int MPI_Request_free(MPI_Request *request)
- Deallocation of the request object does not have any effect on the associated non□blocking send or receive operation.

- Avoiding Deadlocks By using non-blocking communication operations we can remove most of the deadlocks associated with their blocking counterparts.

```
 1    int a[10], b[10], myrank;
 2    MPI_Status status;
 3    ...
 4     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
 5    if (myrank == 0) {
 6      MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
 7      MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
 8    }
 9    else if (myrank == 1) {
10      MPI_Recv(b, 10, MPI_INT, 0, 2, &status, MPI_COMM_WORLD);
11      MPI_Recv(a, 10, MPI_INT, 0, 1, &status, MPI_COMM_WORLD);
12    }
13    ...
```

However, if we replace either the send or receive operations with their non-blocking counterparts, then the code will be safe, and will correctly run on any MPI implementation.

```
 1    int a[10], b[10], myrank;
 2    MPI_Status status;
 3    MPI_Request requests[2];
 4    ...
 5     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
 6    if (myrank == 0) {
 7      MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
 8      MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
 9    }
10    else if (myrank == 1) {
11      MPI_Irecv(b, 10, MPI_INT, 0, 2, &requests[0], MPI_COMM_WORLD);
12      MPI_Irecv(a, 10, MPI_INT, 0, 1, &requests[1], MPI_COMM_WORLD);
13    }
14    ...
```

This example also illustrates that the non-blocking operations started by any process can finish in any order depending on the transmission or reception of the corresponding messages. For example, the second receive operation will finish before the first does.

MPI Collective Communications

Collective Communications Collective Communications

The sending and/ or receiving of messages to/ from groups of processors. A collective communication implies that all processors need participate in the communication.

- Involves coordinated communication within a group of processes
- No message tags used
- All collective routines block until they are locally complete
- Two broad classes :
  - Data movement routines
  - Global computation routines

MPI Collective Communications

Collective Communication

- Communications involving a group of processes.
- Called by all processes in a communicator.
- Examples:
  - Barrier synchronization.
  - Broadcast, scatter, gather.
  - Global sum, global maximum, etc.jj

MPI Collective Communications

Characteristics of Collective Communication

- Collective action over a communicator
- All processes must communicate
- Synchronization may or may not occur
- All collective operations are blocking.
- No tags.
- Receive buffers must be exactly the right size

MPI Collective Communications

Communication is coordinated among a group of processes

- Group can be constructed " **by hand**" with MPI group-manipulation routines or by using MPI topology-definition routines
- Different communicators are used instead
- No non-blocking collective operations

Collective Communication routines - Three classes

- Synchronization
- Data movement
- Collective computation

MPI Collective Communications

Barrier

A barrier insures that all processor reach a specified location within the code before continuing.

- C:

```
int MPI_Barrier (MPI_Comm comm);
```

Broadcast

A broadcast sends data from one processor to all other processors.

- **C**:

```
int MPI_Bcast ( void *buffer, int count, MPI_ Datatype
                datatype, int root, MPI_Comm comm);
```

- MPI_Bcast sends the data stored in the buffer buf of process source to all the other processes in the group.
- The data received by each process is stored in the buffer buf The data that is broadcast consist of count entries of type data type .
- The amount of data sent by the source process must be equal to the amount of data that is being received by each process; i.e., the count and data type fields must match on all processes.

MPI Collective Computations

Global Reduction Operations

- Used to compute a result involving data distributed over a group of processes.
- Examples:
  - Global sum or product
  - Global maximum or minimum
  - Global user-defined operation

MPI Collective Computations

```
int MPI_Reduce (void *sendbuf, void *recvbuf, int
                count, MPI_Datatype datatype , MPI_Op
                op,    int root, MPI_Comm comm) ;
```

- MPI_Reduce combines the elements stored in the buffer sendbuf of each process in the group, using the operation specified in op, and returns the combined values in the buffer recvbuf of the process with rank target.
- Both the sendbuf and recvbuf must have the same number of count items of type datatype.

MPI Collective Computations

Collective Computation Operations

MPI_Name	Operation
MPI_LAND	Logical and
MPI_LOR	Logical or
MPI_LXOR	Logical exclusive or (xor)
MPI_BAND	Bitwise AND
MPI_BOR	Bitwise OR
MPI_BXOR	Bitwise exclusive OR

MPI Collective Computations

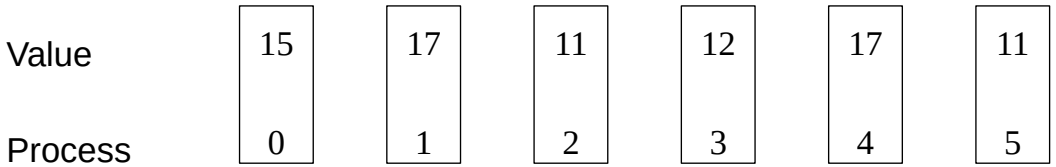
Collective Computation Operation

MPI Name	Operation
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_PROD	Product
MPI_SUM	Sum
MPI_MAXLOC	Maximum and location
MPI_MAXLOC	Maximum and location

Collective Communication Operations

- The operation `MPI_MAXLOC` combines pairs of values (v_i, l_i) and returns the pair (v, l) such that v is the maximum among all v_i's and l is the corresponding l_i (if there are more than one, it is the smallest among all these l_i's).

- `MPI_MINLOC` does the same, except for minimum value of v_i.




```
MinLoc(Value, Process) = (11, 2)
MaxLoc(Value, Process) = (17, 1)
```

An example use of the `MPI_MINLOC` and `MPI_MAXLOC` operators.

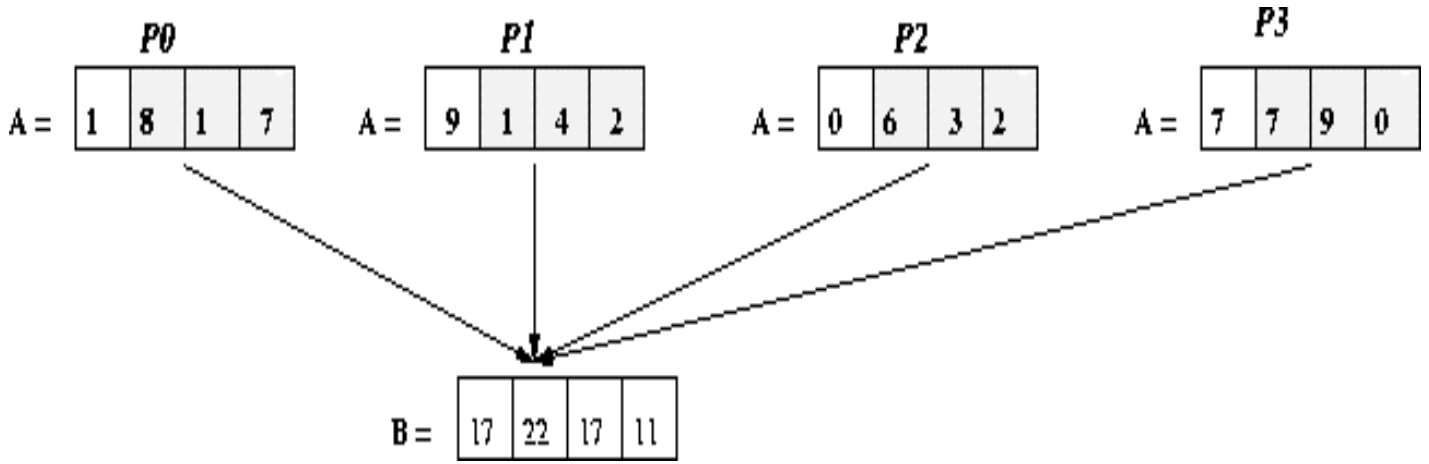
Collective Communication Operations

MPI datatypes for data-pairs used with the MPI_MAXLOC and MPI_MINLOC reduction operations.

MPI Datatype	C Datatype
MPI_2INT	pair of ints
MPI_SHORT_INT	short and int
MPI_LONG_INT	long and int
MPI_LONG_DOUBLE_INT	long double and int
MPI_FLOAT_INT	float and int
MPI_DOUBLE_INT	double and int

Reduction

A reduction compares or computes using a set of data stored on a ll processors and saves the final result on one specified processor.



Global Reduction (sum) of an integer array of size 4 on each processor and accumulate the same on processor P1

MPI_All reduce

MPI provides the MPI_All reduce operation that returns the result to all the processes.

```
int MPI_Allreduce(void *sendbuf, void *recvbuf,
int count,  MPI_Datatype datatype, MPI_Op op,
MPI_Comm comm)
```

Note that there is no target argument since all processes receive the result of the operation.

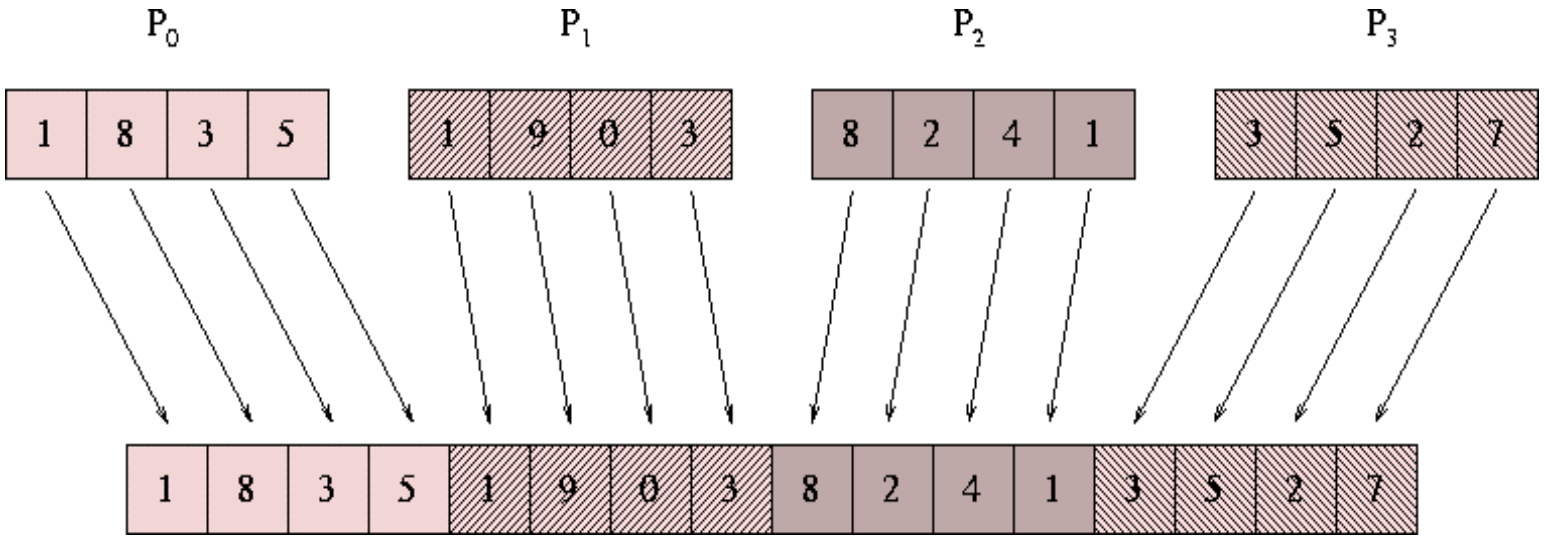
Prefix

int MPI_Scan(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

- MPI_Scan performs a prefix reduction of the data stored in the buffer sendbuf at each process and returns the result in the buffer recvbuf .
- The receive buffer of the process with rank i will store, at the end of the operation, the reduction of the send buffers of the processes whose ranks range from 0 up to and including i .
- The type of supported operations (i.e., op) as well as the restrictions on the various arguments of MPI_Scan are the same as those for the reduction operation MPI_Reduce .

Gather

Accumulate onto a single processor, the data that resides on all processors



Gather an integer array of size of 4 from each processor

Gather

int MPI_Gather(void *sendbuf, int sendcount,
MPI_Datatype senddatatype, void *recvbuf, int recvcount,
MPI_Datatype recvdatatype, int target, MPI_Comm comm)

- Each process, including the target process, sends the data stored in the array sendbuf to the target process.
- As a result, if p is the number of processors in the communication comm, the target process receives a total of p buffers.
- The data is stored in the array recvbuf of the target process, in a rank order.
- That is, the data from process with rank i are stored in the recvbuf starting at location i * sendcount

Gather

MPI also provides the MPI_Allgather function in which the data are gathered to all the processes and not only at the target process.

```
int MPI_Allgather(void *sendbuf, int sendcount,
MPI_Datatype senddatatype, void *recvbuf, int recvcount,
MPI_Datatype recvdatatype, MPI_Comm comm)
```

The meanings of the various parameters are similar to those for MPI_Gather ; however, each process must now supply a recvbuf array that will store the gathered data.

Gather

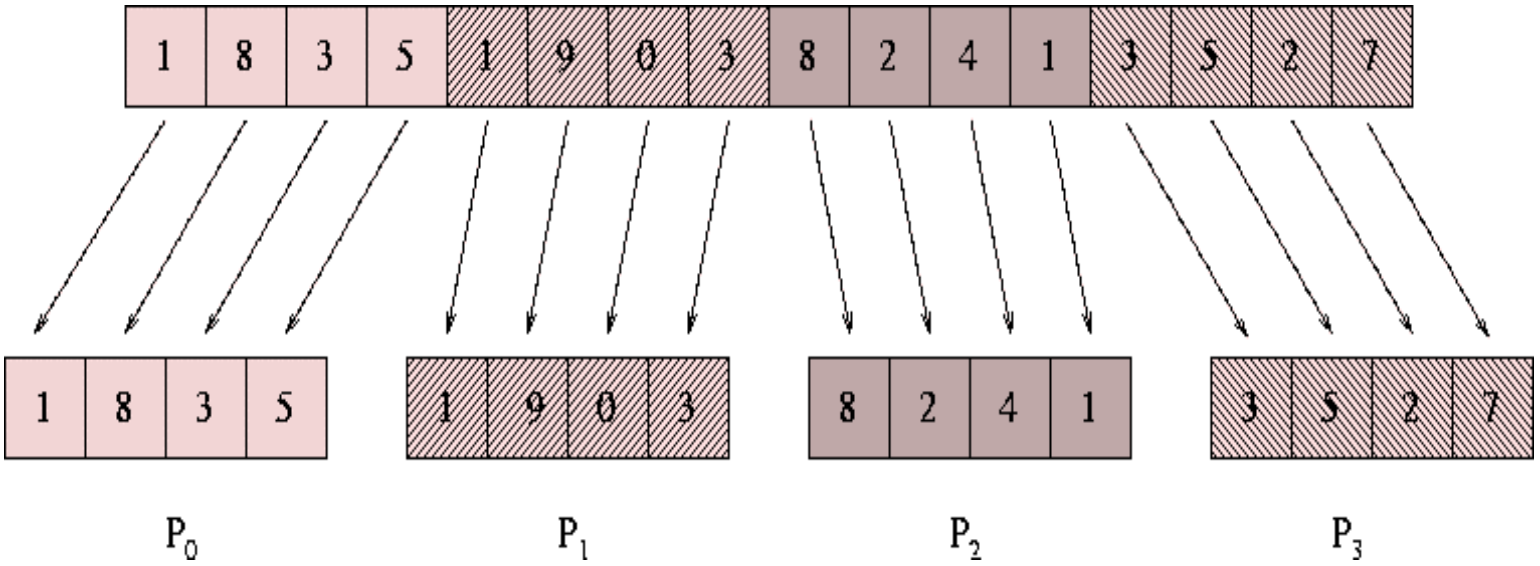
- MPI also provides versions in which the size of the arrays can be different.
- The vector variants of the MPI_Gather and MPI_Allgather operations are provided by the functions MPI_Gatherv and MPI_Allgatherv , respectively.

```
int MPI_Gatherv(void *sendbuf, int sendcount,
MPI_Datatype senddatatype, void *recvbuf, int *recvcounts,
int *displs, MPI_Datatype recvdatatype, int target,
MPI_Comm comm)

 int MPI_Allgatherv(void *sendbuf, int sendcount,
MPI_Datatype senddatatype, void *recvbuf, int *recvcounts,
int *displs, MPI_Datatype recvdatatype, MPI_Comm comm)
```

Scatter

Distribute a set of data from one processor to all other processors.



Scatter an integer array of size 16 on 4 processors

Scatter

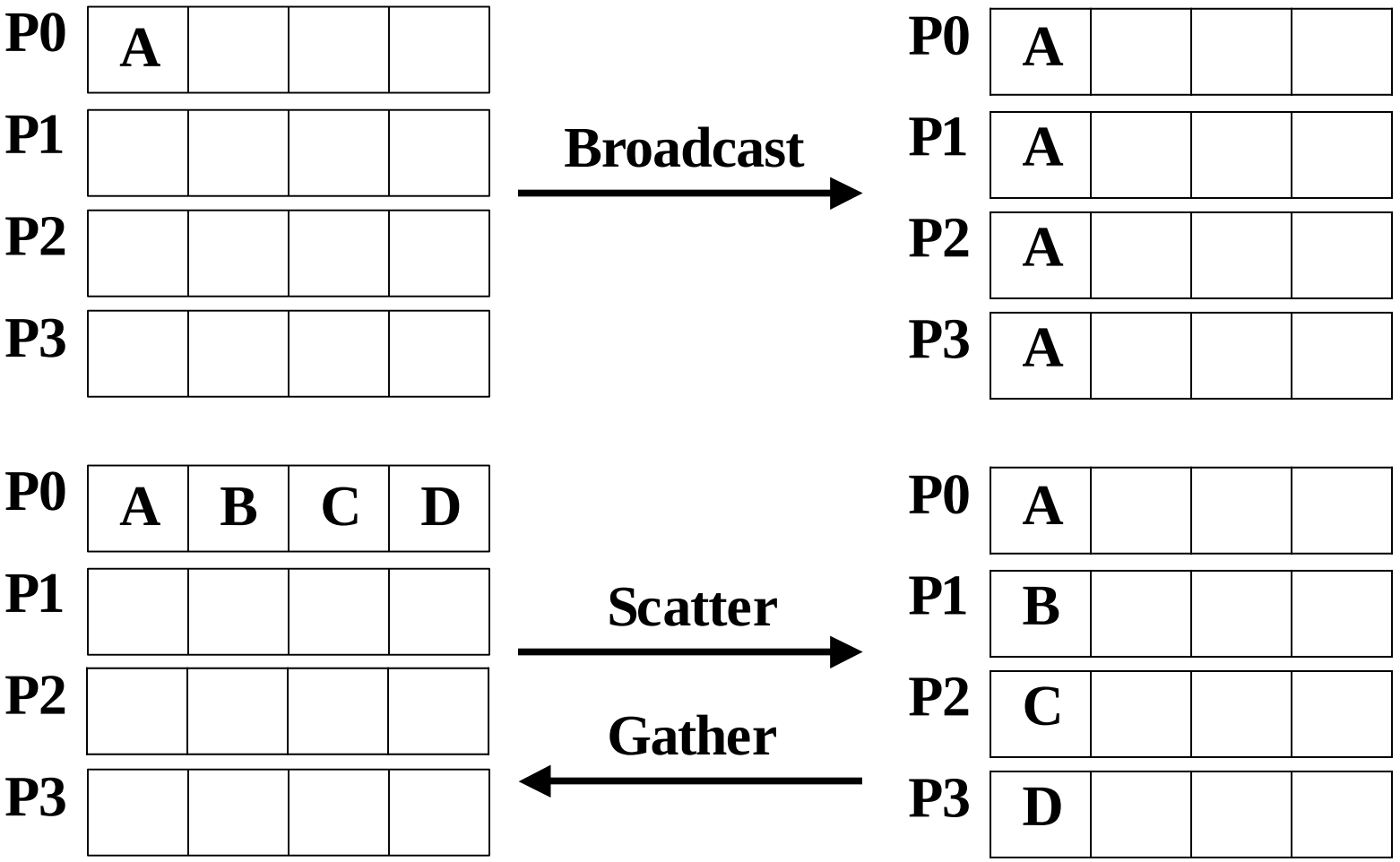
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, int source, MPI_Comm comm)

- The source process sends a different part of the send buffer sendbuf to each processes, including itself.
- The data that are received are stored in recvbuf . Process i receives sendcount contiguous elements of type senddatatype starting from the i * sendcount location of the sendbuf of the source process.
- MPI_Scatter must be called by all the processes with the same values for the sendcount , senddatatype , recvcount , recvdatatype , source , and comm arguments.

Scatter

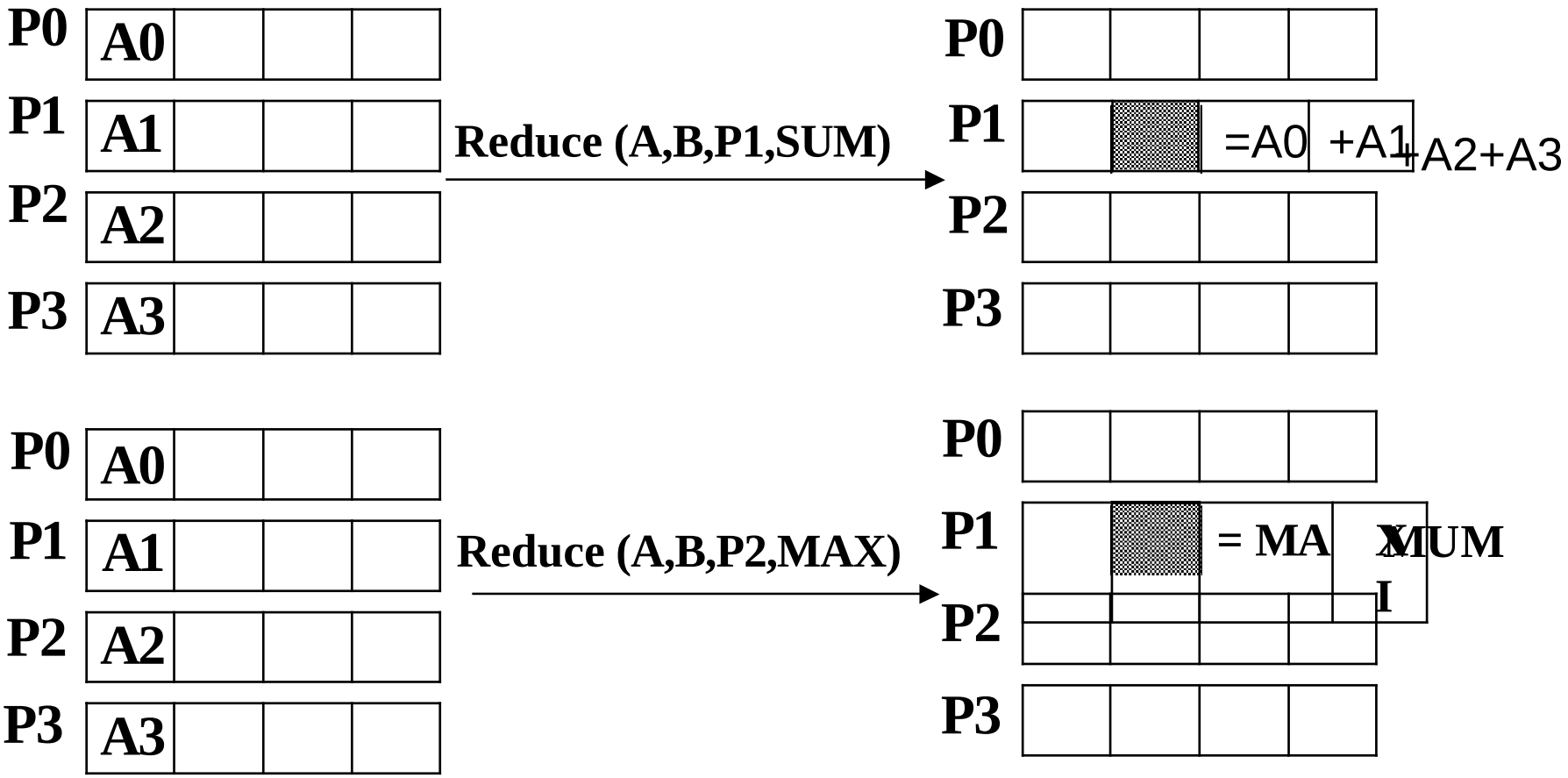
int MPI_Scatterv(void *sendbuf, int *sendcounts, int *displs, MPI_Datatype senddatatype, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, int source, MPI_Comm comm)

MPI Collective Communication



Representation of collective data movement in MPI

MPI Collective Communication



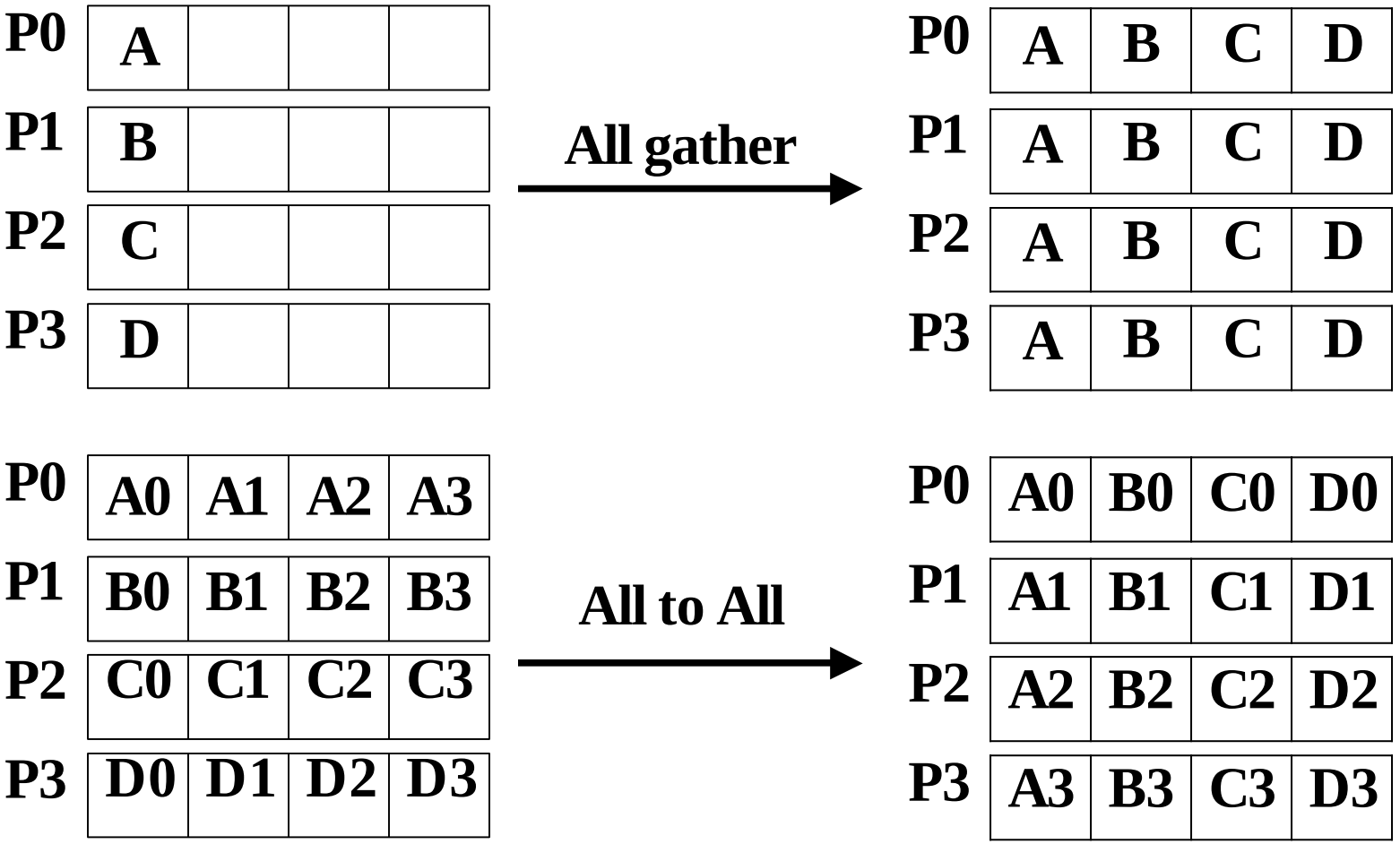
Representation of collective data movement in MPI

MPI Collective Communications & Computations

Allgather	Allgatherv	Allreduce
Alltoall	Alltoallv	Bcast
Gather	Gatherv	Reduce
Reduce Scatter	Scan	Scatter
Scatterv		

- All versions deliver results to all participating processes
- *V* -version allow the chunks to have different non-uniform data sizes (Scatterv, Allgatherv, Gatherv)
- All reduce, Reduce , ReduceScatter, and Scan take both built-in and user-defined combination functions

MPI Collective Communication



Representation of collective data movement in MPI

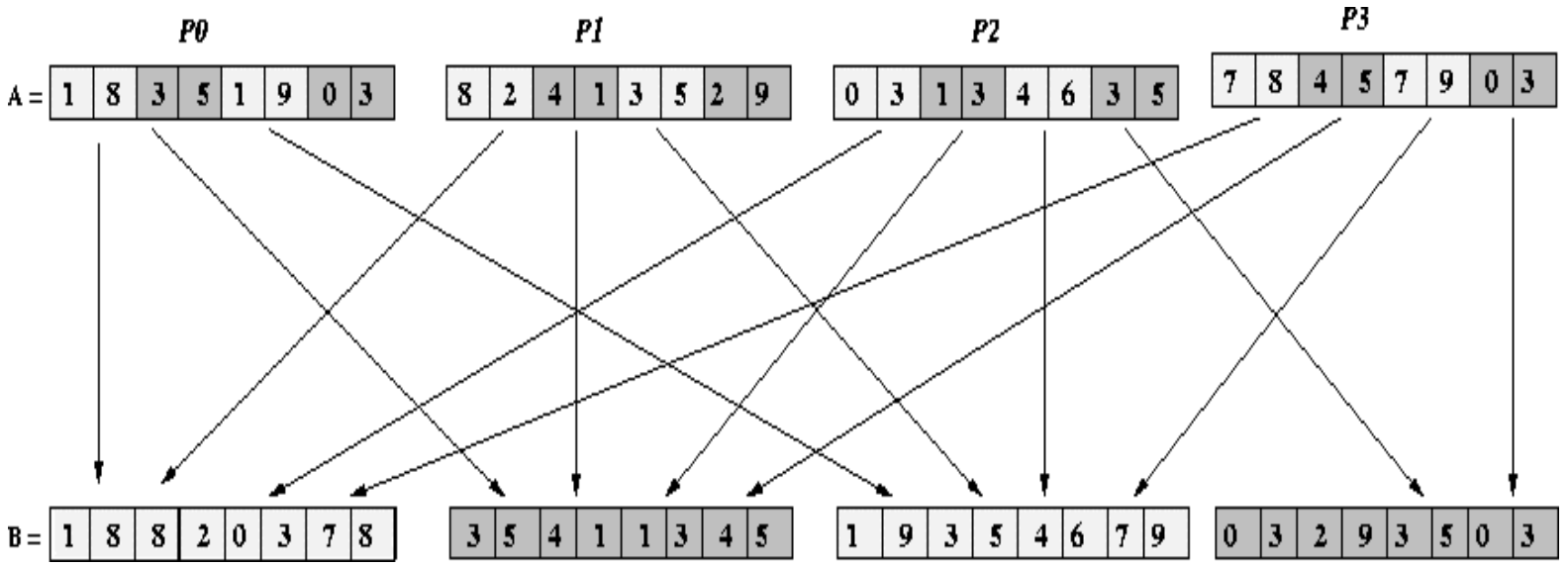
ALL to ALL

int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, MPI_Comm comm)

- Each process sends a different portion of the sendbuf array to each other process, including itself.
- Each process sends to process i sendcount contiguous elements of type senddatatype starting from the i * sendcount location of its sendbuf array.
- The data that are received are stored in the recvbuf array. Each process receives from process i recvcount elements of type recvdatatype and stores them in its recvbuf array starting at location i * recvcount .

All-to-All

Performs a scatter and gather from all four processors to all other four processors. every processor accumulates the final values



All-to-All operation for an integer array of size 8 on 4 processors

Contd..

int MPI_Alltoallv(void *sendbuf, int *sendcounts, int *sdispls MPI_Datatype senddatatype, void *recvbuf, int *recvcounts, int *rdispls, MPI_Datatype recvdatatype, MPI_Comm comm)

- The parameter sendcounts is used to specify the number of elements sent to each process, and the parameter sdispls is used to specify the location in sendbuf in which these elements are stored.
- In particular, each process sends to process i, starting at location sdispls[i] of the array sendbuf, sendcounts[i] contiguous elements.
- The parameter recvcounts is used to specify the number of elements received by each process, and the parameter rdispls is used to specify the location in recvbuf in which these elements are stored.
- In particular, each process receives from process i recvcounts[i] elements that are stored in contiguous locations of recvbuf starting at location rdispls[i]

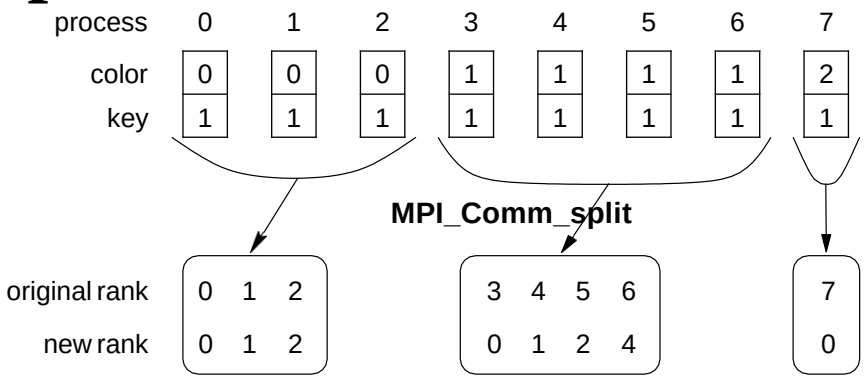
Groups and Communicators

- In many parallel algorithms, communication operations need to be restricted to certain subsets of processes.
- MPI provides mechanisms for partitioning the group of processes that belong to a communicator into subgroups each corresponding to a different communicator.
- The simplest such mechanism is:

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
        MPI_Comm *newcomm)
```

This operation groups processors by color and sorts resulting groups on the key.

Groups and Communicators



Using `MPI_Comm_split` to split a group of processes in a communicator into subgroups.

Groups and Communicators

- In many parallel algorithms, processes are arranged in a virtual grid, and in different steps of the algorithm, communication needs to be restricted to a different subset of the grid.

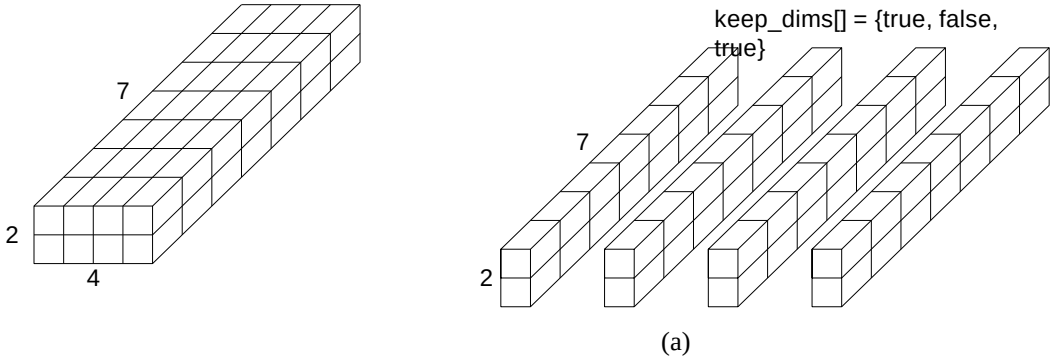
- MPI provides a convenient way to partition a Cartesian topology to form lower-dimensional grids:

```
int MPI_Cart_sub(MPI_Comm comm_cart, int *keep_dims,
        MPI_Comm *comm_subcart)
```

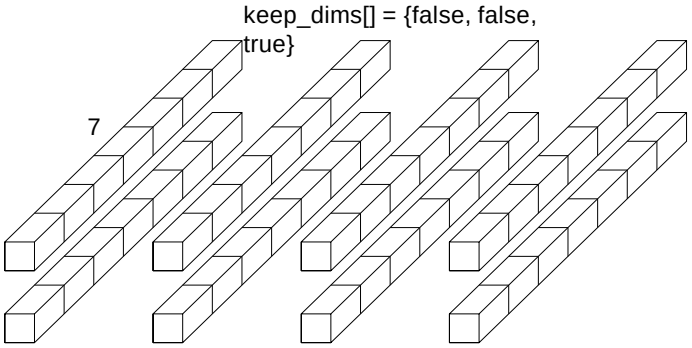
- If `keep_dims[i]` is true (non-zero value in C) then the `i`th dimension is retained in the new sub-topology.

- The coordinate of a process in a sub-topology created by `MPI_Cart_sub` can be obtained from its coordinate in the original topology by disregarding the coordinates that correspond to the dimensions that were not retained.

Groups and Communicators



(a)



(b)

Splitting a Cartesian topology of size 2 × 4 ×7 into (a) four subgroups of size 2 × 1 × 7, and (b) eight subgroups of size 1 × 1 × 7.

CUDA

Instructor
Dr. B Krishna Priya

Outline

- The Age of Parallel Processing
- Central Processing Units
- The Rise of GPU Computing
- A brief history of GPUs
- Early GPU computing, CUDA: What is CUDA architecture
- using the CUDA architecture
- Applications of CUDA
  - Medical Imaging
  - Computational Fluid Dynamics
  - Environmental Science

Agile of Parallel Processing

- 2010- Computers are shipped with multicore central processor. Example:
  - Dual-core
  - low-end netbook machines to 8- and 16-core

    workstation computers- supercomputer/ mainframe
  - Portable Music Player
  - Mobile Phone

Contd..

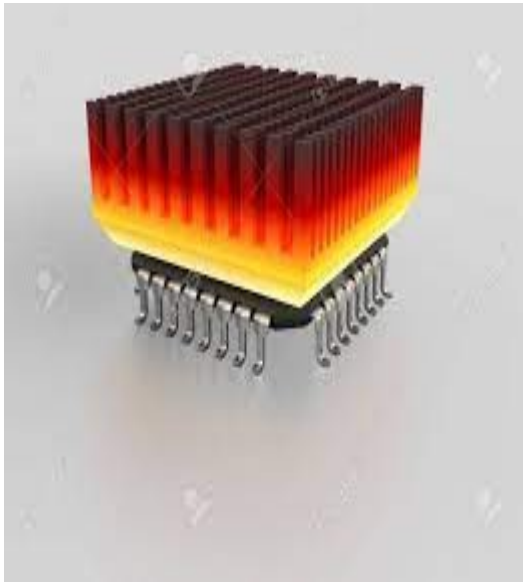
- Software developers will need to cope with a variety of parallel computing platforms and technologies in order to provide novel and rich experiences for an increasingly sophisticated base of users
- Command prompts are out and multithreaded graphical interfaces are in.
- Cellular phones that only make calls are out and phones that can simultaneously play music, browse the Web, and provide GPS services are in.

Central Processing Unit

- 1980s, consumer central processing units (CPUs) ran with internal clocks operating around 1MHz.
- 30 years later- most desktop processors have clock speeds between 1GHz and 4GHz, nearly 1,000 times faster than the clock on the processing original personal computer.
- Increasing the CPU clock speed -performance has been improved, it has always been a reliable source for improved performance.

Contd..

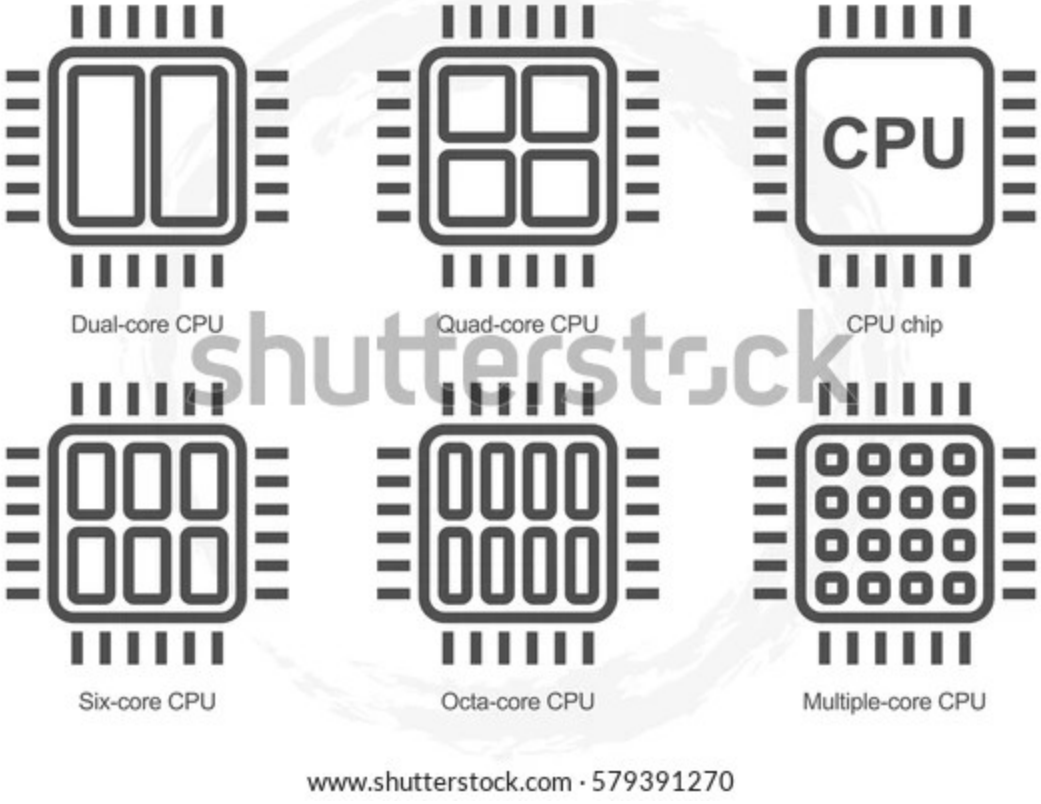
- limitations
  - Limitations of fabrication of integrated circuits.
  - Power and heat restrictions as well as a rapidly approaching physical limit to transistor size.



- Overcome the Limitations

  Personal Computers- three-, four-, six-, and eight-core central processor units

Contd..


CPU
Dual-core CPU
Quad-core CPU
CPU chip
Six-core CPU
Octa-core CPU
Multiple-core CPU
shutterstock
www.shutterstock.com · 579391270

The Rise of GPU Computing

A Brief History of GPU

- In the late 1980s and early 1990s, the growth in popularity of graphically driven operating systems such as Microsoft Windows helped create a market for a new type of processor.
- 1990s-2D display accelerators are used for the personal computers. These display accelerators offered hardware-assisted bitmap operations to assist in the display and usability of graphical operating systems.

Contd..

- 1992- Silicon Graphics opened the programming interface to its hardware by releasing the OpenGL library. Silicon Graphics intended OpenGL to be used as a standardized, platform-independent method for writing 3D graphics applications
- 1992s

1. Doom, Duke Nukem 3D, and Quake helped ignite a quest to create progressively more realistic 3D environments for PC gaming.

2.NVIDIA, ATI Technologies, and 3dfx Interactive began releasing graphics accelerators that were affordable computing enough to attract widespread attention.

Contd.

- NVIDIA’s GeForce 256-transform and lighting computations could be performed directly on the graphics processor, thereby enhancing the potential for even more visually interesting applications.
- NVIDIA’s release of the GeForce 3 series was the computing industry’s first chip to implement Microsoft’s then-new DirectX 8.0 standard.

Early GPU Computing

- The GPUs of the early 2000s were designed to produce a color for every pixel on the screen using programmable arithmetic units known as pixel shaders.
- A pixel shader uses its (x,y) position on the screen as well as some additional information to combine various inputs in computing a final color.

- The additional information could be input colors, texture coordinates, or other attributes that would be passed to the shader when it ran.
- The arithmetic being performed on the input colors and textures was completely controlled by the programmer, researchers observed that these input “colors” could actually be any data
- Inputs were actually numerical data signifying something other than color, programmers could then program the pixel shaders to perform arbitrary computations on this data.
- The results would be handed back to the GPU as the final pixel “color,” although the colors would simply be the result of whatever computations the

CUDA Architecture

- The GeForce 8800 GTX was also the first GPU to be built with NVIDIA’s CUDA Architecture.
- The CUDA Architecture included a unified shader pipeline, allowing each and every arithmetic logic unit (ALU) on the chip to be marshaled by a program intending to perform general-purpose computations.
- NVIDIA intended this new family of graphics processors to be used for general-purpose computing.
- The execution units on the GPU were allowed arbitrary read and write access to memory as well as access to a software-managed cache known as shared memory.

Using the CUDA Architecture

- CUDA C became the first language specifically designed by a GPU company to facilitate general-purpose computing on GPUs.
- A specialized hardware driver to exploit the CUDA Architecture’s massive computational power.

Applications of CUDA

- Medical Imaging
- Computational fluids Dynamics
- Environment science

Introduction to CUDA C

outline

- Introduction to CUDA C
- A First Program, Hello world
- A kernel call
- Passing parameters
- Querying devices
- Using device properties

A First Program

```
#include "../common/book.h"
int main( void )
{
    printf( "Hello, World!\n" );
      return 0;
}
```

-Example illustrates that there is no basic difference between Standard C and Cuda C

- This program runs entirely on Host

A First Program

- Host= CPU + System Memory
- Device= GPU+ its Memory
- A function that executes on the Device is Kernel

Contd..

```
#include<iostream.h>
_ _global_ _void kernel(void) {
}
Int main(void){
kernel<<<1,1>>>{};
printf(“Hello, World\n”);
return 0;
}
```

- The angle brackets denote arguments are passed to the runtime system.
- These are not arguments to the device code but are parameters that will influence how the runtime will launch our device code

Contd..

- CUDA C adds the __global__ qualifier to standard C.
- This mechanism alerts the compiler that a function should be compiled to run on a device instead of the host.
- Example: nvcc gives the function kernel() to the compiler that handles device code, and it feeds main() to the host compiler.
- Host code to one compiler and device code to another compiler

Passing Parameter

- #include "book.h“
- __global__ void add(int a, int b, int *c)
- { *c = a + b;
- }
- int main(void)
- {
- int c; int *dev_c;
- HANDLE_ERROR(cudaMalloc((void**)&dev_c, sizeof(int)));
- add<<1,1>>(2, 7, dev_c);
- HANDLE_ERROR(cudaMemcpy(&c, dev_c, sizeof(int), cudaMemcpyDeviceToHost));
- printf("2 + 7 = %d\n", c);
- cudaFree(dev_c);
- return 0;
- }

Contd.

- Pass parameters to a kernel as done with any C function.
- Allocate memory to do anything useful on a device, such as return values to the host.
- Angle-bracket syntax notwithstanding, a kernel call looks and acts exactly like any function call in standard C.
- The runtime system takes care of any complexity introduced by the fact that these parameters need to get from the host to the device

Contd.

- HANDLE_ERROR(): calls is a utility macro, It simply detects that the call has returned an error, prints the associated error message, and exits the application with an EXIT_FAILURE code.

- cudaMalloc():allocation of memory, but it tells the CUDA runtime to allocate the memory on the device.
- cudaMalloc((void**)&dev_c, sizeof(int))
- The first argument is a pointer to the pointer you want to hold the address of the newly allocated memory, and the second parameter is the size of the allocation you want to make.

Contd..

- The compiler cannot protect you from this mistake, either. It will be perfectly happy to allow dereferences of device pointers in your host code because it looks like any other pointer in the application. We can summarize the restrictions on the usage of device pointer as follows:
- Pass pointers allocated with cudaMalloc() to functions that execute on the device.
- Use pointers allocated with cudaMalloc()to read or write memory from code that executes on the device.
- Pass pointers allocated with cudaMalloc()to functions that execute on the host.
- Cannot use pointers allocated with cudaMalloc()to read or write memory from code that executes on the host

Contd..

- cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);

- cudaMemcpy() :memory on a device through calls to from host code.
- These calls behave exactly like standard C memcpy() with an additional parameter to specify which of the source and destination pointers point to device memory.
- In the example, notice that the last parameter to cudaMemcpy() is cudaMemcpyDeviceToHost, instructing the runtime that the source pointer is a device pointer and the destination pointer is a host pointer

Contd..

- cudaMemcpyHostToDevice would indicate the opposite situation, where the source data is on the host and the destination is an address on the device.
- Finally, we can even specify that both pointers are on the device by passing cudaMemcpyDeviceToDevice.
- If the source and destination pointers are both on the host, we would simply use standard C’s memcpy() routine to copy between them

Querying devices

- cudaGetDeviceCount(): to know how many devices in the system were built on the CUDA architecture. These devices will be capable of executing kernels written in CUDA C.
- int count;
  HANDLE_ERROR(cudaGetDeviceCount(&count));
- cudaDeviceProp structure returns(see from the textbook)

Device Properties

- cudaDeviceProp prop; memset(&prop, 0, sizeof(cudaDeviceProp)); prop.major = 1; prop.minor = 3
- cudaGetDeviceCount() and cudaGetDeviceProperties(), we could iterate through each device and look for one that either has a major version greater than 1 or has a major version of 1 and minor version greater than or equal to 3

Contd..

- cudaChooseDevice() to have the CUDA runtime find a device that satisfies this constraint. The call to cudaChooseDevice() returns a device ID that we can then pass to cudaSetDevice().

```
#include "../common/book.h"

int main( void ) {
    cudaDeviceProp  prop;
    int dev;

    HANDLE_ERROR( cudaGetDevice( &dev ) );
    printf( "ID of current CUDA device:  %d\n", dev );

    memset( &prop, 0, sizeof( cudaDeviceProp ) );
    prop.major = 1;
    prop.minor = 3;
    HANDLE_ERROR( cudaChooseDevice( &dev, &prop ) );
    printf( "ID of CUDA device closest to revision 1.3:  %d\n", dev );
    HANDLE_ERROR( cudaSetDevice( dev ) );
}
```

```
#include <stdio.h>

// Print device properties
void printDevProp(cudaDeviceProp devProp)
{
    printf("Major revision number:         %d\n",  devProp.major);
    printf("Minor revision number:         %d\n",  devProp.minor);
    printf("Name:                          %s\n",  devProp.name);
    printf("Total global memory:           %lu\n",  devProp.totalGlobalMem);
    printf("Total shared memory per block: %lu\n",  devProp.sharedMemPerBlock);
    printf("Total registers per block:     %d\n",  devProp.regsPerBlock);
    printf("Warp size:                     %d\n",  devProp.warpSize);
    printf("Maximum memory pitch:          %lu\n",  devProp.memPitch);
    printf("Maximum threads per block:     %d\n",  devProp.maxThreadsPerBlock);
    for (int i = 0; i < 3; ++i)
        printf("Maximum dimension %d of block:  %d\n", i, devProp.maxThreadsDim[i]);
    for (int i = 0; i < 3; ++i)
        printf("Maximum dimension %d of grid:   %d\n", i, devProp.maxGridSize[i]);
    printf("Clock rate:                    %d\n",  devProp.clockRate);
    printf("Total constant memory:         %lu\n",  devProp.totalConstMem);
    printf("Texture alignment:             %lu\n",  devProp.textureAlignment);
    printf("Concurrent copy and execution: %s\n",  (devProp.deviceOverlap ? "Yes" :
"No"));
    printf("Number of multiprocessors:     %d\n",  devProp.multiProcessorCount);
    printf("Kernel execution timeout:      %s\n",  (devProp.kernelExecTimeoutEnabled
?"Yes" : "No"));
    return;
}

int main()
```

```
    printf("Total constant memory:          %lu\n",  devProp.totalConstMem);
    printf("Texture alignment:              %lu\n",  devProp.textureAlignment);
    printf("Concurrent copy and execution: %s\n",  (devProp.deviceOverlap ? "Yes" :
"No"));
    printf("Number of multiprocessors:     %d\n",  devProp.multiProcessorCount);
    printf("Kernel execution timeout:      %s\n",  (devProp.kernelExecTimeoutEnabled
?"Yes" : "No"));
    return;
}

int main()
{
    int devCount;
    cudaGetDeviceCount(&devCount);
    printf("CUDA Device Query...\n");
    printf("There are %d CUDA devices.\n", devCount);

    for (int i = 0; i < devCount; ++i)
    {
        // Get device properties
        printf("\nCUDA Device #%d\n", i);
        cudaDeviceProp devProp;
        cudaGetDeviceProperties(&devProp, i);
        printDevProp(devProp);
    }
    return 0;
}
```

```
CUDA Device Query...
There are 1 CUDA devices.

CUDA Device #0
Major revision number:         3
Minor revision number:         5
Name:                          Tesla K20m
Total global memory:           5032706048
Total shared memory per block: 49152
Total registers per block:     65536
Warp size:                     32
Maximum memory pitch:          2147483647
Maximum threads per block:     1024
Maximum dimension 0 of block:  1024
Maximum dimension 1 of block:  1024
Maximum dimension 2 of block:  64
Maximum dimension 0 of grid:   2147483647
Maximum dimension 1 of grid:   65535
Maximum dimension 2 of grid:   65535
Clock rate:                    705500
Total constant memory:         65536
Texture alignment:             512
Concurrent copy and execution: Yes
Number of multiprocessors:     13
Kernel execution timeout:      No
```

Parallel Programming in CUDA C

Instructor
Dr. B. Krishna Priya

Outline

- CUDA parallel programming
- Summing vectors
- A fun example

CUDA Parallel Programming

Summing Vectors:

- Imagine having two lists of numbers where we want to sum corresponding elements of each list and store the

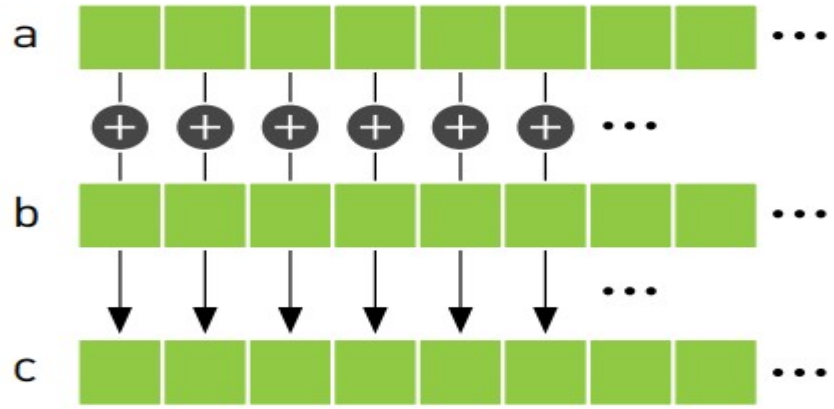


Figure 4.1 Summing two vectors

```c
#include "../common/book.h"

#define N   10

void add( int *a, int *b, int *c ) {
    int tid = 0;    // this is CPU zero, so we start at zero
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 1;   // we have one CPU, so we increment by one
    }
}

int main( void ) {
    int a[N], b[N], c[N];

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }

    add( a, b, c );
```

```
    // display the results
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }

    return 0;
}
```

Most of this example bears almost no explanation, but we will briefly look at the `add()` function to explain why we overly complicated it.

```
void add( int *a, int *b, int *c ) {
    int tid = 0;    // this is CPU zero, so we start at zero
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 1;   // we have one CPU, so we increment by one
    }
}
```

```
void add( int *a, int *b, int *c ) {
    for (i=0; i < N; i++) {
        c[i] = a[i] + b[i];
    }
}
```

- one core initialize the loop with tid = 0 and another with tid = 1. The first core would add the even-indexed elements, and the second core would add the odd indexed elements.

CPU CORE 1

```
void add( int *a, int *b, int *c )
{
    int tid = 0;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 2;
    }
}
```

CPU CORE 2

```
void add( int *a, int *b, int *c )
{
    int tid = 1;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 2;
    }
}
```

GPU vector sums

```
#include "../common/book.h"

#define N   10

int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }
```

```
    // copy the arrays 'a' and 'b' to the GPU
    HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int),
                              cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int),
                              cudaMemcpyHostToDevice ) );

    add<<<N,1>>>( dev_a, dev_b, dev_c );

    // copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int),
                              cudaMemcpyDeviceToHost ) );

    // display the results
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }

    // free the memory allocated on the GPU
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );

    return 0;
}
```

You will notice some common patterns that we employ again:

- We allocate three arrays on the device using calls to `cudaMalloc()`: two arrays, `dev_a` and `dev_b`, to hold inputs, and one array, `dev_c`, to hold the result.
- Because we are environmentally conscientious coders, we clean up after ourselves with `cudaFree()`.
- Using `cudaMemcpy()`, we copy the input data to the device with the parameter `cudaMemcpyHostToDevice` and copy the result data back to the host with `cudaMemcpyDeviceToHost`.
- We execute the device code in `add()` from the host code in `main()` using the triple angle bracket syntax.

Moving on, our `add()` routine looks similar to its corresponding CPU implementation:

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;    // handle the data at this index
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;    // handle the data at this index
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

- N as the number of parallel blocks.
- Collection of parallel blocks a grid.
- This specifies to the runtime system that we want a one-dimensional grid of N blocks (scalar values are interpreted as one-dimensional).
- These threads will have varying values for blockIdx.x, the first taking value 0 and the last taking value N-1.
- Ex: imagine four blocks, all running through the same copy of the device code but having different values for the variable blockIdx.x.
- See the figures: Actual code being executed in each of the four parallel blocks looks like after the runtime substitutes the appropriate block index for blockIdx.x

BLOCK 1

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 0;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 2

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 1;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 3

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 2;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 4

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 3;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

- HANDLE_ ERROR() macros that we’ve sprinkled so liberally throughout the code will detect and alert you to the situation.